HANDBOOK

Introduction

The Hobby Module lets you program your own games on your advanced programmable video system and opens up a whole new world of interest. You may write your own programs and store them on an audio cassette recorder (not included) for permanent storage.

This manual will teach you how to write your own programs. If you have no previous experience you may find chapters 5 and 6 difficult. If so we suggest you read the other chapters then try some of the examples and come back to the detailed explanation when you have grasped the idea.

Note - The Hobby Module must be used together with the special mains adaptor supplied. Do not use the smaller mains adaptor which was originally supplied with the game console.

I N D E X

Section

1 HOW IT WORKS
- 1.1. Microprocessor
- 1.2. Number Storage
- 1.3. Memory Types
- 1.4. Hexadecimal Notation

2 SIMPLE PROGRAMMING AND THE MONITOR
- 2.1. Entering and Controlling Programs
- 2.2. Getting Started

3 THE PROGRAMMABLE VIDEO INTERFACE AND PERIPHERALS
- 3.1. Scanning the Screen
- 3.2. Objects
- 3.3. Multiple Objects
- 3.4. Scores
- 3.5. Background
- 3.6. Collisions
- 3.7. Joysticks
- 3.8. Sound
- 3.9. Keyboards
- 3.10. Writing to the Screen

4 THE BASIC INSTRUCTION SET

5 SOME PROGRAMMING TECHNIQUES
- 5.1. Multi-Byte Arithmetic
- 5.2. Two's Complement Arithmetic
- 5.3. Condition Codes
- 5.4. Logical Function
- 5.5. Indirect Addressing
- 5.6. Relative Addressing
- 5.7. Subroutines
- 5.8. Interrupts
- 5.9. Instruction Timing

Section

6 ADVANCED INSTRUCTIONS

6.1. Relative Addressing
6.2. Logical Functions
6.3. Compare
6.4. Conditional Branching
6.5. Test Under Mask
6.6. Rotate Register
6.7. Decimal Adjust
6.8. Program Status Words

7 EXAMPLES OF PROGRAMMING

7.1. Setting Up Objects and Background
7.2. Joystick Control, Moving Objects, Wait Loops, Bank 1
7.3. Multiple Objects
7.4. The Survival Game
7.5. P.V.I. - Art

8 Figures

1 Memory Contents
2 Example of PVI Object
3 Locations for Specifying the Background

9 Appendices

1 Condensed Monitor Instructions
2 PVI Locations
3 Microprocessor Instruction Codes
4 Glossary
5 Monitor Subroutine Summary
6 Conversion Tables
7 Connecting a Cassette Recorder

10 References

1. HOW IT WORKS

1.1 Microprocessors

Video games are controlled by a highly complex electronic circuit called a microprocessor. This device can manipulate numbers - moving them from one place to another, adding them together, etc. - according to a very basic set of rules. It forms the basis (the central processing unit, or C.P.U.) of a wide variety of systems, from pocket calculators, vending machines, and cash registers, to car engine controllers and Teletext controllers.

To organise the basic set of rules into a useful system the microprocessor needs a list of instructions to follow. This list is called its program, and in your video games unit this program is contained in the plug-in module. By changing to another module, and therefore another program, a completely different game is selected. Some programs are written to give the player a choice of games in one module, by choosing between alternative blocks of program (known as sub-routines - which will be discussed later on), within the main program, but the player cannot make his own rules or invent his own games.

The Hobby Module has been designed to let you write your own programs, both for the satisfaction of seeing your own ideas working, and for playing games of your own invention.

The program must be written in a language which the microprocessor understands, and this means using only the instructions listed later in this manual, known as the instruction set of the microprocessor (a Signetics 2650). As the circuit understands only numbers, not words, the program consists of a list of numbers, each being a code for an instruction. As an illustration, if we had an instruction set 1 = turn right, 2 = turn left, 3 = drive forward, 4 = drive backwards, 5 = open garage, 6 = stop, then 5,4,2,6,3,6 would get the car to the road, and 3,2,6,5,3,6 would put it back in the garage.

Writing a program using these code numbers is possible but difficult; to make the program more meaningful a set of mnemonics is used to represent the numeric code. These are known as 'assembly code', and the numeric code as 'object code'. When a program has been written in assembly code it must be changed ('assembled') into object code before it can be entered into the module; this is often done on a small computer, but a pencil and the conversion table at the back of this manual are quite adequate. Programming sheets are provided for this purpose.

When a microprocessor is used in a minicomputer, or one of the 'personal' or 'home' computers now becoming popular, a high level language is used to communicate with the machine, such as BASIC, FORTRAN or PASCAL. These languages look much more like English, but require extensive programs (compilers or interpreters)to translate each high level instruction into a set of object code instructions before it is obeyed. This is relatively slow; programs in object code are faster and more efficient.

1.2 Number Storage

Before examining the instruction set in detail we need to understand some aspects of how the microprocessor works, particularly the way it remembers ('stores') numbers.

The electronic circuits used are of the switching type, which can exist in only one of two states, 'on' or 'off' (this refers to the signal coming out of the circuit). These states can be used to represent the numbers 0 and 1, but higher numbers cannot be represented in one circuit. In the decimal system, which humans normally use for calculations, when we run out of figures we 'carry' a 1 into another digit, so 9 + 1 (each one digit) = 10 (2 digits). Similarly 99 + 1 = 100 (3 digits) etc. The electronic system has to carry when the count is at 1, as it cannot represent 2, so 1 + 1 = 10, 11 + 1 = 100. This is the binary system. The decimal numbers 0, 1, 2, 3, 4, 5, 6, 7 therefore become 0, 1, 10, 11, 100, 101, 110, 111 in binary code. These binary numbers can be stored in a bank of switching circuits, each representing one digit, so three circuits, storing three binary digits, can represent 000 to 111 (binary) which is 0 to 7 (decimal). Incidentally the term 'binary digit' is used so frequently it is shortened to 'bit' for convenience.

The usual method of storing numbers in microprocessors is to use blocks of eight circuits, storing eight bits, which can contain the numbers B'0000 0000' to B'1111 1111', which is D'0' to D'255', using the convention B'' and D'' to indicate binary and decimal forms. Banks of circuits for this purpose are known as 'registers', and a binary number containing eight bits is known as a 'byte' (making 4 bits, or half a byte, a 'nibble').

An eight bit byte in a register can therefore represent any number up to D'255', but it can be used for other purposes; for example a 0 or 1 in each individual bit in the register could be used to record the presence or absence of an object on the screen, a record of whether a particular object has been hit in a shooting game, or whether a particular card has been played in a card game. To identify individual bits in a byte the eight bits are numbered from 0 to 7, where bit 0 is the least significant (right hand end) bit.

It was explained earlier that instructions to the microprocessor are in the form of numbers. These comprise one or more eight bit binary numbers. The program therefore consists, as far as the microprocessor is concerned, of a list of 8 bit bytes held in a set of registers. (Think of a very tall bookcase with room for eight books on each shelf). The microprocessor starts by reading the first byte, translating it into an instruction which it obeys, then reading the next byte, etc. This set of registers is called 'program memory'.

1.3 Memory Types

In the video games modules the contents of the program memory is permanent (provided it is not ill-treated) whereas in the hobby module the program can be typed into its memory from the keypads, or fed in from a tape recorder. There are therefore two types of memory, the permanently programmed type, Read-Only Memory, or ROM (so called because one cannot "write" into it) and the type which can be written into, Random Access Memory, or RAM (so called because one can read from or write to any register in the memory). In RAM the contents are lost when the supply is turned off, so when you have written your program record it on tape before switching off the unit.

The microprocessor contains seven RAM type registers, known as the working registers, to contain numbers involved in its current calculations. These are arranged, for technical reasons, so that only four can be accessed immediately. These are known as R0, R1, R2, and R3. If it is necessary to use more than these four, one bit in a special register called the Program Status Word (described later) can be changed to a '1'; R0 will then still be accessible, but R1 to R3 will now refer to three different registers. These two sets of R1 to R3 are called Bank 0 and Bank 1. Example 2 shows a use of these two banks.

In all but the simplest programs more than seven numbers are used, and extra RAM is needed to hold numbers while the c.p.u. is working on other tasks - for example while the microprocessor is using its registers to calculate the position of one badminton player it needs a temporary store for the positions of the other player and the ball. This temporary storage is often referred to as 'scratchpad', and is accessed in exactly the same way as program RAM, indeed the same RAM maybe used for both purposes at different times.

We now have several types of register. The microprocessor contains working registers, can read from program ROM, can read or write program in RAM, and store numbers in scratchpad RAM. To avoid confusion from now on we shall use the word 'register' only for the working registers inside the microprocessor and use the term 'memory location' for the other types.

1.4 Hexadecimal Notation

The microprocessor works with 8 bit numbers, such as B'1011 0110' or B'11000111'. These appear confusing to the human brain, which would prefer the decimal equivalents, D'182' and D'199'. Converting from binary to decimal is a tedious chore, so a compromise system has been adopted.

If we split the 8 bit byte into two 4 bit nibbles, then each nibble has a maximum value of B'1111' or D'15'. If we could represent all numbers up to 15 by one digit then the 4 bit nibble would be directly equivalent to one digit. The decimal numbers D'10' to D'15' are therefore represented by the letters A to F, giving the conversion:

4 bit binary	0000	0001	0010	0011	0100	0101	0110	0111
Decimal	0	1	2	3	4	5	6	7
Hexadecimal	0	1	2	3	4	5	6	7

4 bit binary	1000	1001	1010	1011	1100	1101	1110	1111
Decimal	8	9	10	11	12	13	14	15
Hexadecimal	8	9	A	B	C	D	E	F

As the highest digit is equivalent to D'15' the value of the second digit is 16 (i.e. we 'carry' 16) and the system is hexadecimal (or 'base 16'). This type will be marked as H' ' where necessary.

Representing each 4 bit nibble as a single digit means that a direct conversion from one to the other is simple - the conversion can be memorised with a little practice, and the programmer is urged to THINK HEXADECIMAL; do not convert to decimal numbers.

As an illustration, as H'B' = B'1011' and H'6' = B'0110', then B'1011 0110' = H'B6', a much simpler conversion than D'182'.

The system can be extended to longer binary numbers. For example, the microprocessor selects which memory location it wishes to access by transmitting a binary number on 15 wires (known as a 15 bit address bus because it carries the address of the desired location to all parts of the system - fig. 1). This 15 bit number has a maximum value of B'111 1111 1111 1111', which when divided into 4 bit nibbles from the right hand end, is immediately seen to be H'7FFF'. Conversion to D'32,767' takes much longer.

2. SIMPLE PROGRAMMING AND THE MONITOR

2.1. Entering And Controlling Programs

In ordinary video games the players communicate with the microprocessor using the keypads and joysticks. To use the hobby module we still use the keypads, but in a different way; the two keypads are placed side by side and used as a single keyboard. When the hobby module is plugged in the new functions of the keys become:

(left keypad)			(right keypad)		
RCAS	WCAS	C	D	E	F
BP1/2	REG	8	9	A	B
PC	MEM	4	5	6	7
-	+	0	1	2	3

SYSTEM KEYS

START
SELECT
RESET
ON/OFF

These are shown on the overlays provided.

The microprocessor has two separate functions when the hobby module is used. First it allows the programmer to enter his program into memory (the RAM at locations 0900 to 0FFF), to examine the contents of memory, etc. In this mode it is running under the control of a program in ROM in locations 0 to 07FF, known as the monitor program. When the reset key is pressed the microprocessor always returns to this mode. Secondly the microprocessor runs the program which the programmer has entered if it is started at location 0900 (or later). In this mode the monitor program is not active, and the keypads can have any function the user program dictates.

2.2. Getting Started

Having switched on your unit as normal and plugged in your Hobby Module, press RESET. You now have ++++ at the bottom left of the t.v. screen.

Let us now put in a small program starting at address 0900. Note that when you press a key a bleep sounds. This will show up keying errors.

Press 'MEM' then 9, 0, 0. The screen should now show Ad = 900. Press '+' and you will see 0900 HH where HH are two hex characters. We are looking at the contents of address H900 and if now we press keys 0, then 4, we have entered our first byte of code at address H'900'. Now press 06 and a new line will be seen: 0901 06. Continue with 05, 02, 81, C2, 40 (the 40 should be in address 906). We have entered our first program. Press '+' again and we see that the line 0906 did not change and a new line 0907 has appeared. Pressing '+' will increment the addresses and '-' decrement them without change so that we can step forward or back at will and examine or change the contents of any location. Note that after each line the screen contents move up one line - this is known as 'scrolling' and will be discussed later.

Just before we run our little program let us do one further thing. Press BP and the screen will say BP1 = 0000; type in a new address 0906 and press '+' to enter it into the monitor. We have now put a 'Breakpoint' in our program and when we run it, the program will 'Break' as the processor reaches address 906, returning to monitor level and saving all the registers so that if we wish we can continue the program from the Breakpoint as if nothing had happened.

Pressing PC displays the current Program Counter contents (the location from which the next instruction will be taken). This is normally 900, unless a Breakpoint has been met, but can be changed if required. Press '+' to start the program. The screen should now say BP1 = 0906 which means the processor has found our Breakpoint and returned to monitor level. Now if you press REG you will see in R0 the value that the program has put in it (08). We could now examine/alter any register or memory address and restart the program from the breakpoint address (or any other address if we alter PC). This 'Breakpoint' facility is extremely useful in finding out why a program is not working (which happens to us all) since we can find how far the program runs correctly by running it to progressive BP addresses and checking at each break that we have correct values in our registers and memory addresses.

The monitor will allow us to set up to two breakpoint addresses (by pressing BP again after setting the first one); this helps to check which way a branch statement goes by setting a breakpoint at each of the alternative addresses. Now press 'PC' and you will see it has been changed to 906 so if you press '+' the program will continue from where it stopped. To restart the program from the beginning again retype the start address in PC, and press '+'.

Note that both the PC and Breakpoint addresses should ALWAYS be the address of the first byte of an instruction. In our simple program (see listing on following pages) 900, 902, 904, 905, 906 are fine but 901 or 903 would not work. Incidentally when the monitor displays B or D it does it as lower case letters (b or d) to avoid confusion with 8 or 0.

Let us have a look at the results of our little program. Press 'REG' and you will see R0 = 08. Press '+' and we see R1 = 02; press '+' again and we have R2 = 08. Now that is good since our program loaded 6 into R0, 2 into R1, added R1 to R0, put the result into R0 and then stored R0 into R2. At that point the 'Break' occurred at address 906 and the processor returned to the monitor program.

Now press '-' twice and you will see R0 = 08. Press '02' and '+' and we have put a new value (02) into R0. If we ran the program again, by re-inserting the breakpoint at 906 (which is cleared to 0 when the program arrives at it), resetting the Program Counter PC = 0902 and pressing '+', we should get a new answer in R2 of R0 + R1 = 4. Try this for yourself and check the register contents.

If you press 'REG' in monitor and press '+' repeatedly you will get registers R0 to R8. We have only used R0 to R3 which are known as 'Bank 0' registers. R4 to R6 are available only when 'Bank 1' is set by a program instruction. (R0 is common to both banks and R4 to R6 on the display are called R1 to R3 of Bank 1 in programming). R7 and R8 are the Program Status Word which we will consider later.

The program we put in is shown below, laid out formally. When you write a program you should do it like that so that you can always find your way round it later; it's all too easy to look at a jumble of code and wonder what it did! Some printed programming sheets are supplied at the back of this manual.

Assembly Code Label	Instruction	Address	Object Code	Comment
BEGIN	LODI,R0 06	0900	04 06	Value 6 in R0
	LODI,R1 02	902	05 02	Value 2 in R1
	ADDZ R1	904	81	Add R1 to R0
	STRZ R2	905	C2	Answer into R2
	HALT	906	40	Stop

You should start by writing the complete assembly code then go back to the beginning and put in the addresses and instruction code (i.e. assemble the program) line by line so that you keep track of the address. If your program is complicated it helps to begin with a flow diagram before writing the assembly code; this is a logical description of what the program should do.

Labels (such as "BEGIN" in the above program) are a temporary substitute for the address of an instruction which can be calculated when the program is assembled. Any words, numbers etc., can be used (e.g. START, LOOP, DATA7 etc). The coded instructions, however, are in a fixed mnemonic code, a shorthand for a detailed instruction; (it also makes it look impressive to those who do not know about it). "LODI,R0 06" is short for "Load immediately into register zero the hex number 6". You can do the second one. "ADDZ R1" means "Add to register zero the contents of R1" (the original contents of R0 will be lost but R1 will stay the same). "STRZ R2" means "Store the contents of R0 into R2" and "HALT" means "Stop" of course. You should normally use a HALT instruction at the end of your program, otherwise, after the processor has executed the last instruction you put in, it will move to the next address and try to execute the random contents: you can get some very funny effects that way.

There are only two further facilities to learn about in the monitor program and they are RCAS and WCAS. When you get your program (or part program) working properly it is very useful to store it on an ordinary cassette recorder. This saves you the trouble of typing it in again.

Connect the cassette recorder to the socket on the Hobby Module (appendix 7) and insert a good quality cassette. Press WCAS and the monitor will ask you for the addresses at the beginning and end of your program, and the location at which the program starts, by displaying 'BEG=', 'END=' and 'SAD='. Insert the addresses, each followed by the '+' key. 'FIL' will then appear (a program on tape is called a file); enter a single hex digit (except 0) and, before pressing the '+' key, start the tape recorder in the 'record' mode. The program will then be recorded.

To read the program back from cassette press the 'RCAS' key and enter the number of the file when requested, followed by '+'. Start the cassette player in 'play' mode; the microprocessor will then wait for the file with that number to be played back, and will read it into the correct memory locations. Note that while it is reading the file two dots flash on the screen. If it finds a file with a different number the dots flash at double speed and the number is displayed.

It is good practice when recording programs to record a spoken comment before the program, including the title, file number, beginning and end addresses and the address at which the program should be started, especially if this is other than 0900.

After a program is recorded it remains in the memory (until the power is switched off), and it is good practice to verify the recording. This is done by rewinding the cassette, and proceeding as for reading the cassette, except that '-' instead of '+' is pressed after the file number.

If you do not enter a file number when asked but simply press + (or -) the monitor will read (or verify) the first file it finds.

When the microprocessor is reading, verifying or writing a file it displays the last line of text. At completion a full screen of text returns. After a successful 'read' operation the start address is displayed so that the '+' key starts the program at that location. If an error is found during 'read' or 'verify' reading is stopped and an error indicated. As the file is held on the tape in blocks of 16 bytes the microprocessor displays the number of the block in which the error is detected.

A condensed list of monitor functions is given in Appendix 1 at the back of the manual.

Before we leave this section, there is one further point concerning programming. This processor system has built into it a number of 'Interrupt' sources. An 'Interrupt' is a signal to the processor from outside which tells it to stop what it is doing and attend to the 'Interrupt'. This will be dealt with in detail later in the book but we need to know how to ignore interruptions for our first few programs.

In order to ignore 'Interrupts' your program should begin with the two instructions below.

Label	Instruction	Address	Code	
BEGIN	BCTA,UN PROG	900	1F 09 04	Jump round address 903
	RETC,UN	903	17	Stop further Interrupts
PROG	- - - - - -	904	- - - -	Program Start

PROG is the real start of your program and the effect of the above will become clear when you read the sections on Interrupts and RETC/RETE. You should use the above otherwise your programs may not work at all, particularly if address 903 is not the start of an instruction, and you will also find RESET must be pressed several times to stop your program running.

3. THE PROGRAMMABLE VIDEO INTERFACE AND PERIPHERALS

3.1. Scanning The Screen

So far we have discussed only the microprocessor and its associated memory. In the games unit there is another circuit called the Programmable Video Interface (PVI). The television picture is drawn 50 times in each second. To provide the right information to the screen at the right time the microprocessor would continuously be taking numbers from memory and passing them, bit by bit, to the display electronics. The PVI does this automatically, leaving the microprocessor free to do more complex tasks. In addition it contains some RAM, the contents of which are interpreted in special ways which will be described in the following sections. This memory is addressed in the same way as other memory (see memory map, figure 1), either by the program or by the MEM key function. However, if the MEM key is used to read a PVI RAM location in the range 1FC0 to 1FCB it resets it to zero. (Try 'MEM', '1FC0', '+', then START to get back to normal).

3.2. Objects

In the majority of games 'objects', such as animals, vehicles, balls, or letters and numbers are shown on the screen, and often moved. The PVI has four blocks of RAM in locations 1F00 to 1F0E, 1F10 to 1F1E, 1F20 to 1F2E and 1F40 to 1F4E, dedicated to showing four objects. In each block the first ten bytes are used to provide a rectangle of 8 x 10 squares which can be used to draw the object by writing a 1 (object colour) or 0 (background colour) in each bit. An example is shown in figure 2. Locations 1FnA and 1FnC (where n = 0, 1, 2 or 4 for the four objects) define the horizontal and vertical position of the top left hand corner of the object on the screen, with a range of 0 (left side) to B0 (right side) and 0 (top) to FE (bottom). You cannot alter these with the MEM key, because the monitor program uses the objects to write messages on the screen, so it immediately changes them back again if you try.

In addition to the four objects, duplicates of them may be shown on the screen by entering into location 1FnB the horizontal position of the duplicates (which will all have the same horizontal position, i.e. will be vertically under each other), and in location 1FnD one less than the desired separation of the duplicates (figure 2).

A choice of four sizes is available for each object independently, according to the following code:

0	0	smallest
0	1	x2
1	0	x^4
1	1	x^8

This is entered in memory location 1FC0, using bits 1 and 0 for object 1, 3 and 2 for object 2 etc., for example 0001 1011, or H'1B' would set objects 1 to 4 to x^8, x^4, x2 and x^1 respectively. This size choice also scales the duplicates of the objects.

The colours of the objects are set in location 1FC1 (objects 1 and 2) and 1FC2 (objects 3 and 4) according to the codes:

	bit c	bit b	bit a
White	0	0	0
Yellow	0	0	1
Magenta	0	1	0
Red	0	1	1

	bit c	bit b	bit a
Cyan	1	0	0
Green	1	0	1
Blue	1	1	0
Black	1	1	1

where c, b and a are:

1FC1	-	-	c	b	a	c	b	a
			Object 1			Object 2		
1FC2	-	-	c	b	a	c	b	a
			Object 3			Object 4		

For example, 1FC1 = 00001010 = H'0A', 1FC2 = 00011100 = H'1C' sets objects 1 to 4 to yellow, magenta, red and cyan respectively.

If an object is not wanted either all bits may be set to 0 (which is done by the START key, saving program space), or the horizontal co-ordinate may be set to 'FF' (location 1FnA). The duplicate can be deleted by setting the horizontal offset to FF (location 1FnB).

3.3. Multiple Objects

If we set up an object size x1 with duplicates at an offset of H'10' we can have up to nine objects down the screen. The PVI writes these out to the screen, one after the other, and on completion of each object it sends a signal to the microprocessor. The microprocessor responds by executing a special program which must be written at location 0903 (this will become clear when you read the sections on subroutines and interrupts in the INSTRUCTION SET chapter). If, at this location, we write a program which changes the contents of the object memory, then when the PVI writes out the next "duplicate" it will be a different object. In this way each of the four objects can produce several different objects, giving a maximum of 64 objects on the screen.

We can change the position of the objects, but as they are "duplicates" the vertical and horizontal co-ordinate object locations (1FnA and 1FnC) will not work - the horizontal co-ordinate must be put in the duplicate co-ordinate location (1FnB) and the vertical co-ordinate changed by using the offset location (1FnD). Of course subsequent "duplicates" will be offset from each preceding "duplicate", so increasing one offset also moves down subsequent "duplicates".

If more than one object is treated in this way it is necessary to decide when an interrupt is signalled, which object has just been completed and is ready for re-writing. Bits 0 to 3 in location 1FCA are used for this purpose, indicating objects 4 to 1 respectively. The same signal is used when the screen sweep is complete, so that the first object can be restored ready for the next sweep. This is identified by bit 6 in location 1FCB (known as Vertical Reset Leading Edge, VRLE).

An example of multiple object generation is given in example 3 but a thorough reading of the "INSTRUCTION SET" chapter will be required before this can be understood.

3.4. Scores

A facility is provided for showing scores during a game. Four digits are displayed, the values being set in the four nibbles of locations 1FC8 and 1FC9 (values of H'A' and above blank out the digit). Bit 0 of location 1FC3 sets the score display either to the top of the screen (0) or the bottom (1) and bit 1 gives two 2-digit pairs (0) or one 4-digit number (1). The colour cannot be programmed but is automatically set to the complement of the background colour.

3.5. Background

As well as the four objects and their duplicates, it is possible to put on the picture a variety of lines and rectangles to make up a background; for example a football pitch can be marked out, a set of irregular 'islands' can be put on a map, or a maze shown. As with all memory locations the values can be changed during the program, so the background facility can be used to "fill" tubs, build houses etc.

The background covers the part of the screen between H'14' and H'DC' vertically, H'20' and H'A0' horizontally. Within this area the screen is divided into H'14' horizontal strips, which are alternately narrow and wide (in the ratio 1:9), shown in figure 3. Each horizontal strip is divided into H'10' sections, each of which is individually controlled by the contents of memory locations 1F80 to 1FA7. For example, the left 8 sections of the third strip down are controlled by the eight bits in location 1F84 and the right eight sections by the eight bits in location 1F85; so 1F84 = B'10000000' = H'80' and 1F85 = B'00000001' = H'1' would produce a dot at each side of the screen.

A '1' in any of these bits produces a small bar of the height of the strip, and 1/8 of the width of the section. This width can be extended using locations 1FA8 to 1FAC. Each of these locations controls four horizontal strips, 1FA8 controlling the top 4 strips etc. In each location bits 7 and 6 are used to extend all the bars in those 4 strips, if they are turned on, according to the code:

bit 7	bit 6	bar width
0	0	1/8
0	1	1/4
1	0	1/8
1	1	1/2

The other 6 bits of each byte are used so that a '1' overrides the setting in bits 7 and 6, extending the bars in one strip to full section width, i.e. giving a continuous bar when adjacent sections are turned on. As each byte controls two narrow and two wide strips, and 6 bits are available, each wide strip is divided into two for bar widening purposes, and the following code used:

bit 0 extends bars in the top narrow strip of the set of four

bit 1 extends bars in the upper half of the upper wide strip

bit 2 extends bars in the lower half of the upper wide strip

bit 3 extends bars in the lower narrow strip

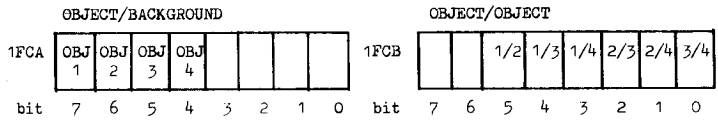
bit 4 extends bars in the upper half of the lower wide strip

bit 5 extends bars in the lower half of the lower wide strip

The colours of the screen (bits 2, 1 and 0) and of the background (bits 6, 5 and 4) are controlled by location 1FC6, with the complementary colour code to the objects. Bit 3 of 1FC6 turns on (1) the background and screen colours, or sets both to black (0).

3.6. Collisions

In ball games, car racing and similar games, collisions can be monitored between objects (cars, people, balls) and either other objects or the background (road markers, maze walls, tennis court lines etc). Location 1FCB contains information on inter-object collisions and 1FCA object-background collision, as follows:



3.7. Joysticks

So that the joysticks can be used to control the game, a circuit is included to measure the position of the joysticks, and record the result as a binary number in locations 1FCC (left) and 1FCD (right). However, as the measurements of the joystick positions occur during the t.v. picture scan, the numbers in these locations are correct for only a limited period during each 1/50th of a second. Bit 6 of location 1FCB (VRLE) is used to mark when the information is valid. This can either be used in a "wait loop" to delay the program until the correct time to sample the joystick position (as in example 3), or an interrupt routine can be used, as we discussed under 'multiple object generation'. As the end of the screen sweep (VRLE is 1) is used to cause an interrupt the joystick measurement and multiple object reset can be included in the same subroutine.

Each joystick has two co-ordinates, forwards and lateral. If the command '7640' is given at VRLE, the readings at the next VRLE of locations 1FCC and 1FDD will give the forward position and similarly the command '7440' will give the lateral position (example 3).

3.8. Sound

The contents of memory location 1FC7 controls the pitch of the note transmitted to the speaker of the television set. A value of zero turns off the note, while a value between 2 and H'40' gives a note whose frequency can be calculated from the formula $n = \frac{7.8125}{f} - 1$, where f is the frequency in kHz, and n the required number in decimal notation.

Some values are:

n(hex)	note	n(hex)	note	n(hex)	note	n(hex)	note
03	B	0D	C sharp	18	E flat	2C	F
04	G	0E	C	1A	D	2E	E
05	E	0F	B	1B	C sharp	31	E flat
06	C sharp	10	B flat	1D	C	34	D
07	B	11	A	1F	B	37	C sharp
08	A	12	G sharp	21	B flat	3B	C
09	G	13	G	23	A		
0A	F	14	F sharp	25	G sharp		
0B	E	15	F	27	G		
0C	D	17	E	29	F sharp		

The contents of memory location 1E80 control a random noise generator, an explosion circuit and a four level volume control.

The bits have the following functions activated by writing a 1 in the appropriate bit.

Bit No.	Function
0	— — —
1	— — —
2	Enable normal P.V.I. sound
3	Mix P.V.I. sound with random noise
4	Start explosion (≥20msec)
5	Identical colours background/screen/objects
6	Volume control 1
7	Volume control 2

Bits 6 and 7 set the volume as follows:

00 = High, 10 = Medium-high, 01 = Medium-low, 11 = Low

Examples:

Value H'04' at location H'1E80' will enable the P.V.I. sound at maximum volume.

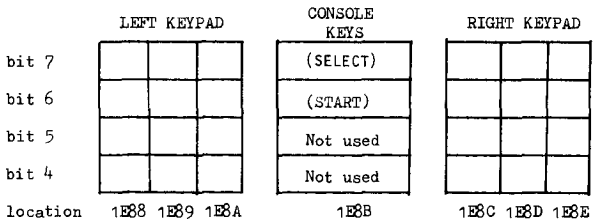
Value H'08' will mix P.V.I. sound with noise at lowest volume level.

Value H'10' during at least 1 frame period (20msec) will trigger the explosion circuit.

3.9. Keyboards

We have looked at the keyboard as a way of communicating with the monitor program. It can also be read during games programs, and there are two ways to do this.

The keys are directly accessible in locations 1E88 to 1E8E, as follows:-



To test for a particular key - say centre top of the right pad - a typical program would be:

```
LODA,R0 1E8D        0C 1E 8D    Select key column
TMI,R0 80           F4 80       Test key
BSTA,A1 RSHOOT      3C HH HH    Subroutine if pressed
LODA,R0 1E89        0C 1E 89    Repeat for left pad
TMI,R0 80  etc.
```

In the monitor program a subroutine is available which gives several advantages. It tests keys over a period of 5 screen sweeps (100msecs) to eliminate 'contact bounce' in the switches, and gives '2 key rollover' which means that if a second key is pressed before the first is released, the second one will not be recognised until the first is released.

The subroutine KEYSCN is called by an instruction BSTA,UN KEYSCN (3F 01 4E).

To use the subroutine, a location (89F) called 'keyboard status' or MKBST is used. If, during the interrupt subroutine, bit 7 of this location is cleared and the subroutine called the bit will be set to 1 if a key is pressed, and a 5 bit code put into the lower five bits, as in the following table:

LEFT KEYPAD			CONSOLE KEYS	RIGHT KEYPAD		
03	07	0B	0F	13	17	1B
02	06	0A	0E	12	16	1A
01	05	09	Not used	11	15	19
00	04	08	Not used	10	14	18

The program to read this could be:

```
START   BCTA,UN MAIN        900   1F 0A 00    Jump to main program
        TPSU SENSE          903   B4 80       Interrupt subroutine
        RETE,N1             905   36          Return if not screen
                                                flyback
        LODA,R0 MKBST       906   0C 08 9F
        ANDI,R0 7F          909   44 7F       Mask off bit 7
        STRA,R0 MKBST       90B   CC 08 9F    and store back
        BSTA,UN KEYSCN      90E   3F 01 4E    Keyboard subroutine
        LODA,R0 MKBST       911   0C 08 9F    Re-examine MKBST
        BSTA,N KEY          914   3E 09 50    Subroutine if key
                                                pressed
        RETE,UN             917   37
```

Note that in location 914 to 916, as we are testing for bit 7; its presence makes the byte look like a negative number so a 'branch if negative' instruction can be used.

In location 950 (or any convenient location), called in locations 914 to 916, a subroutine would then be written to deal with the key.

If it is desired to treat the keyboards separately, locations 89D (RKBST - right keyboard) and 89E (LKBST - left keyboard) can be used instead of MKBST, with the same coding as above; the console keys are only accessible in MKBST.

3.10. Writing To The Screen

We have seen that the monitor can write 6 lines of 8 characters on the screen. It uses four subroutines to do this which can be accessed from the program.

The screen can be cleared by a subroutine at location 575, by the instruction BSTA,UN CLSCR (3F 05 75) which can be used at any time to clear the P.V.I. object memory.

The characters are displayed by the OUTPUT subroutine at location 0057. This takes bytes from locations 800 to 88F and writes them to the object memory. In order to set up these locations correctly the CONVRT subroutine (below) can be used, but if you want to use symbols other than those available in CONVRT use a program (not MEM) to set up the locations (table below). The four P.V.I. objects and their duplicates are used, with two characters in each object, using the first six bytes of each object only (1Fx0 to 1Fx5, where x = 0,1,2 or 4) and their position registers.

OBJECT1	OBJECT2	OBJECT3	OBJECT4	
800 to 805	806 to 80B	80C to 811	812 to 817	1st Line
818 to 81D	81E to 823	824 to 829	82A to 82F	2nd Line
830 to 835	836 to 83B	83C to 841	842 to 847	3rd Line
848 to 84D	84E to 853	854 to 859	85A to 85F	4th Line
86D to 865	866 to 86B	86C to 871	872 to 877	5th Line
878 to 87D	87E to 883	884 to 889	88A to 88F	6th Line

The standard characters are shown in the next table. To use these, store the character codes into locations 890 to 897, then call subroutine CONVRT at location 1DO by a BSTA,UN CONVRT (3F 01 DO) instruction. This converts the character codes from location 890 to 897 (known as LINE, LINE+1, etc.) into the appropriate bytes and stores them into location 800 to 88F ready for the OUTPUT subroutine.

The codes and characters are:

code	00	01	02	03	04	05	06	07	08
char.	0	1	2	3	4	5	6	7	8

code	09	0A	0B	0C	0D	0E	0F	10	11
char.	9	A	b	C	d	E	F	G	L

code	12	13	14	15	16	17	18	19
char.	I	n	P	r	=	space	+	-

Having set up the locations 800 to 88F by one of these two methods, we can write them to the screen. This is done by the OUTPUT subroutine, by altering the contents of the four objects as their multiples are written to the screen. This is done by timing, not by interrupts, so the display must be synchronised to the subroutine. This is also done automatically if the OUTPUT subroutine is used. In the monitor program the coordinates are set up for giving six lines, and this setting can be used by entering the subroutine at 0057.

The coordinates used are:

	OBJ1	OBJ2	OBJ3	OBJ4
Object horizontal	1F	3F	5F	7F
Multiple horizontal	1F	3F	5F	7F
Object vertical	10	10	10	10
Multiple offset	FF	FF	FF	FF

(FF is effectively - 1)

If other settings are used the subroutine must be entered at location 006F. For example the bottom line above may be used by setting the object vertical coordinates to A0, the characters may be spaced unequally by altering the object horizontal coordinates etc. Offsets should not be changed. A timing loop may be needed to synchronize the lines, set up experimentally.

Using the CONVRT subroutine enters data into the lowest of the six lines, so the coordinates should be arranged to include this line on the screen.

To fill lines other than the lowest the system used is to enter characters into the lowest line, then use subroutine SCROLL to move the line up to the required position. The command BSTA,UN SCROLL (3F 02 3A) moves each line up, losing the top line. Alternatively the subroutine can be entered at 028C, with R1 set according to the number of lines to be scrolled. For example R1 = D0 will scroll 3 lines up, retaining the top three and losing the fourth.

The appropriate settings are:

lines scrolled	1	2	3	4	5
R1 set to	0	E8	D0	B8	A0

After the SCROLL subroutine the sixth line will be unchanged, but may be cleared with the CONVRT subroutine, as 890-89F are cleared.

If you fill locations 800 onwards directly instead of using CONVRT, SCROLL can still be used to move lines up after which new data should be entered into the lowest line in locations 878 to 88F.

The following example shows the use of CONVRT, OUTPUT and SCROLL. Keys are decoded, entered into the lowest line on the screen and scrolled upwards when the line is full. The previous example is used to access the keyboard.

```
BCTA,UN MAIN          900   1F 0A 00
TPSU SENSE            903   B4 80
RETE,N1               905   36
BSTA,UN OUTPUT        906   3F 00 57      Call output subroutine
LODA,RO MKBST         909   0C 08 9F
ANDI,RO 7F            90C   44 7F         Clear MKBST bit 7
STRA,RO MKBST         90E   CC 08 9F
BSTA,UN KEYSCN        911   3F 01 4E      Call keyscan subroutine
LODA,RO MKBST         914   0C 08 9F
BSTA,N KEY            917   3E 09 50      Subroutine if key pressed
RETC,UN               91A   17
```

```
KEY     ANDI,R0 1F          950   44 1F      Mask key code
        LODA,R1 0FFE        952   0D 0F FE   Examine scratchpad
                                               counter
        STRA,R0 890,R1      955   CD 68 90   Storekey in LINE
        ADDI,R1 1           958   85 01      Increment counter for
        COMI,R1 9           95A   E5 09      next LINE byte
        BCTA,LT UPDTE       95C   1E 09 64   End of line ?
        BSTA,UN SCROLL      95F   3F 02 8A   If so - scroll 6 lines
        LODI,R1 0           962   05 00      Reset counter
UPDTE   STRA,R1 0FFE        964   CD 0F FE   Update counter
        BSTA,UN CONVRT      967   3F 01 D0   Convert new character
        RETC,UN             96A   17


MAIN    PPSU,II             A00   76 20      Inhibit interrupt
        CPSL,ALL            A02   75 FF
        PPSL,COM            A04   77 02      Prepare logical
                                               comparison
        LODI,R0 0           A06   04 00
        STRA,R0 0FFE        A08   CC 0F FE   Reset LINE counter
LOOP    CPSU,II             A0B   74 20      Enable interrupt
        BCTR,UN LOOP        A0D   1B 7C      Wait for interrupts
```

4. THE BASIC INSTRUCTION SET

4.1. System

In the examples in the introduction we have seen the general system of handling data. The number being worked on is loaded into one of the working registers (if it is not already there), an arithmetic operation (add, subtract, rotate or test) can then be carried out on it, after which it may be stored into RAM, or left in the register for further operation.

The mnemonic code is in three parts. (1) a three letter code indicates the operation - load into register (LOD), store from register (STR), add to or subtract from register (ADD, SUB) or branch (BCT, BCF, BRN etc). (2) a single letter indicates the source or destination of data; this is known as the addressing mode. Data can be moved between register 0 and the other registers (R1, R2 and R3); this is the 'register zero' mode (Z). Note that R1 to R3 cannot communicate directly with each other. The data may also be supplied directly by the program - for instance "load the number 6 into register 2"; this is the immediate mode (I). Finally the data can be taken from or stored into memory location, in which case the address of the location is required from the program. This is the absolute mode (A). The relative mode (R), which saves program space but is more difficult to decode, will be dealt with later, when more advanced instructions are dealt with. In the first set, which we have called the basic set, are the frequently used instructions which are adequate to write reasonable programs. Other instructions from the advanced set can be picked up as you need them.

These basic instructions will now be listed formally, with their object code equivalents and meanings.

4.2. Load Register Zero (LODZ Rn) : 00 + n

Loads register zero from register n (1 to 3). The previous contents of R0 are lost, register n is unchanged.

Example: LODZ R3 = 03. Loads R0 from R3.

N.B. Do not use code 00 which is indeterminate.

4.3. Load Immediate (LODI,Rn HH) : 04 + n

Loads register n (0 to 3) with HH where HH means two hexadecimal digits, i.e. an 8 bit binary number. The previous contents of Rn are lost.

Example: LODI, R2 17. (06 17). Loads hex 17 into R2.

4.4. Load Absolute (LODA,Rn HH HH) : 0C + n

Loads register n (0 to 3) with the contents of address HH HH. The previous contents of Rn are lost.

Example: LODA,R0 0B 19 (0C 0B 19). Loads R0 with the contents of address B19.

4.5. Store Register Zero (STRZ Rn) : C0 + n

Stores the contents of R0 in Rn (n = 1 to 3). The previous contents of Rn are lost, R0 is unaltered.

Example: STRZ R1 (C1). Stores R0 in R1.

N.B. The code C0 is NO OPERATION (NOP) i.e. do nothing. This is useful for providing spare locations in case extra instructions are needed later or creating a small delay in a program.

4.6. Store Absolute (STRA,Rn HH HH) : CC + n

Stores the contents of Rn (n = 0 to 3) in address HH HH. The contents of Rn are unchanged, the contents of the address are replaced.

Example: STRA,R0 09 20 (CC 09 20). Stores R0 in address 920.

4.7. Add To Register Zero (ADDZ Rn) : 80 + n

Adds the contents of Rn (n = 0 to 3) to the contents of R0. The 8 bit sum replaces the contents of R0. Rn is unchanged.

Example: ADDZ R2 (82). Adds R2 to R0 and puts the sum in R0.

4.8. Add Immediate (ADDI,Rn HH) : 84 + n

Adds HH to Rn (n = 0 to 3) and puts the result in Rn. The contents of Rn are replaced by the 8 bit sum.

Example: ADDI,R3 24 (87 24). Replaces R3 with the 8 bit sum of R3 and hex 24.

4.9. Add Absolute (ADDA,Rn HH HH) : 8C + n

Adds the contents of address HH HH to Rn (n = 0 to 3) and puts the 8 bit sum in Rn. The contents of Rn are replaced by the sum.

Example: ADDA,R2 0970 (8E 09 70). Replaces R2 with the 8 bit sum of R2 and the contents of 970.

4.10. Subtract From Register Zero (SUBZ Rn) : A0 + n

The contents of Rn (n = 0 to 3) are subtracted from the contents of R0 and the result is put into R0. The original contents of R0 are lost.

Example: SUBZ R1 (A1). Subtracts R1 from R0 and puts the result in R0.

4.11. Subtract Immediate (SUBI,Rn HH) : A4 + n

Subtracts HH from Rn (n = 0 to 3) and puts the result into Rn. The original contents of Rn are lost.

Example: SUBI,R3 23 (A7 23). Subtracts hex 23 from R3 and puts the result in R3.

4.12. Subtract Absolute (SUBA,Rn HH HH) : AC + n

Subtracts the contents of address HH HH from Rn (n = 0 to 3) and puts the result into Rn. The original contents of Rn are lost.

Example: SUBA,R0 0B C3 (AC 0B C3). Subtracts the contents of address BC3 from R0 and puts the result in R0.

4.13. Branch On Register Non Zero Absolute

(BRNA,Rn HH HH) : 5C + n

This instruction causes the contents of register n (n = 0 to 3) to be tested for a non-zero value. If the value is non-zero the processor will branch to and execute the instruction at the address HH HH. If the register value is zero the processor will execute the next instruction.

Example: BRNA,R3 0C 07 (5F 0C 07). The processor will branch to address C07 only if R3 is non-zero. If R3 = 0 the next instruction will be executed.

4.14. Branch Unconditionally (BCTA,UN HH HH) : 1F HH HH

This is a special case of an advanced instruction which forces the processor to go unconditionally to the address HH HH and continue program execution from that location onwards.

Example: BCTA,UN 0E 00 (1F 0E 00). The processor will branch to address E00 and execute the instruction at that address.

4.15. Indexing

Indexing is a special form of absolute addressing which is extremely useful. Suppose we take as example the instruction LODA,R0 0B 00. That would load R0 with the contents of address B00. Now let us index it with R3: LODA,R0 0B 00, R3. The instruction now tells the processor to load R0 from address B00 added to the number held in R3 e.g. if R3 holds hex 20,R0 would be loaded from B00 + 20 = B20.

We can make indexing even more useful by indexing with increment or decrement which means that before the register is added to the address the register is incremented (or decremented) by 1. Let us look at a program which illustrates the use of indexing.

```
        LODI,R3   20
LOOP    LODA,R0   0B 00,R3,-      (Indexed with R3 decrementing)

        STRA,R0   1F 00,R3        (Indexed only)

        BRNA,R3   LOOP            (Branch if R3 is not zero)
```

These four lines transfer 32 (H'20') bytes of data from addresses 0B 1F - 0B 00 to addresses 1F 1F - 1F 00. The processor would go round the loop decrementing R3 each time until R3 was zero when it would not jump to LOOP but carry on with the next instruction.

Indexing is coded by adding one of three hex values to the most significant DIGIT of the address (3rd CODE DIGIT). Indexing only = 6, indexing with decrement = 4 and indexing with increment = 2. For example

```
LODA,R0  0B 00,R3,-   =  0F 4B 00   and

STRA,R0  1F 00,R3     =  0F 7F 00
```

Only R0 can be used as the source or destination register and indexing can be applied to any absolute address instruction except branch instructions.

5. SOME PROGRAMMING TECHNIQUES

Before you look at the advanced instruction there are several things you need to know about.

5.1. Multi-Byte Arithmetic

We have seen that an eight-bit number can count up to H'FF' which is D'255'. If we want to go further we must use more than one byte. When the microprocessor adds a number to the contents of a register, if the answer is more than H'FF' (e.g. H'84' + H'96' = H'11A'), then the '1' which carries out of the addition sum is stored in a special location called the CARRY BIT. This location is one bit in a special register in the microprocessor called the PROGRAM STATUS WORD, which contain several similar markers, or STATUS BITS as they are usually called. Another of these bits is the 'WITH CARRY BIT'.'. If this is set to 1 by the instruction PPSL WC (7708) then the CARRY BIT will be added when an addition is called for. If we have a program:

```
ADDI,R0     10
PPSL        WC
ADDI,R1     10
CPSL        WC
```

then the carry bit from the first addition is added into R1; we can now count up to a binary number of 16 bits, which is H'FFFF' or D'32,768'. Note that we clear the WC bit after the addition by a CPSL instruction; it might cause errors if we accidentally added the carry bit in the next sum.

The CARRY BIT is also useful in another way. If we use single byte arithmetic and add, as before, 84 + 96 = 11A, then our register will contain 1A (in decimal, 132 + 150 = 26) which is obviously wrong. If we test the CARRY bit after an addition it will warn us when this error occurs.

5.2. Two's Complement Arithmetic

Processors do not have the facility to handle + or - signs so when they do arithmetic there is a convention that the most significant (i.e. left hand end) bit of a number indicates the sign of the number, 0 for +ve and 1 for -ve. Because of this -ve numbers are held as 'two's complement' numbers which are obtained in the following way.

Suppose we want the two's complement of the binary number 00111100 (Decimal 60). First invert the number, writing a 1 for a 0 and 0 for a 1, i.e. 1100 0011 (this is known as the one's complement). Now add 1 to it, i.e. 1100 0100. This is the two's complement of our original number and it has the interesting property that when added to another number it behaves as the negative value of our number. To prove this let us add it to our original number and we should get 0:

```
      0011 1100
      1100 0100
     ----------
(1)   0000 0000
     ==========
```

Yes, it came to 0, but you will notice that an unwanted 1 fell off the left hand end, which means it went into the CARRY BIT. This is why we reset the WC bit after our two-byte arithmetic example - it could cause errors. If we do a two-byte sum with signed numbers, only the most significant bytes are signed - the others are ordinary 8 bit numbers, but in 2's complement form. If this isn't clear, try a few numbers out - you will find it works.

As the most significant bit is the sign bit, eight bits now give 7F(D'127') to 80(D'-128') instead of 0 to FF(D'255'). If we now add, say, 70(D'114') to 16 (D'22') we get 86(D'-122'), because the carry from the seventh bit added into the sign bit, which is obviously nonsense. Another bit in the Program Status Word, the OVERFLOW BIT, is provided so that this can be detected, just as we saw the CARRY BIT being tested in unsigned 8 bit arithmetic.

5.3. Condition Codes

There are two bits held in the Program Status Word which are called the CONDITION CODE bits (CC1 and CC2) and whenever a register in the processor is changed the condition code bits are affected. In addition they are affected by the COMPARE, TEST UNDER MASK, ROTATE and DAR instructions.

Whenever a LOAD, ADD, SUBTRACT, STORE REGISTER ZERO, AND, IOR or EOR instruction is executed the new contents of the changed register affect the condition code in the following way.

Register Contents	Condition Code
Positive	01
Zero	00
Negative	10

For the condition codes after a COMPARE or an Advanced instruction see the relevant instructions.

You will need to use condition codes when you use some of the BRANCH or RETURN instructions. For example if we want to use the instruction BRANCH ON CONDITION TRUE to an absolute address (BCTA) we would write BCTA, con HH HH. Here 'con' is the condition we want to be true and we can have our condition either Zero (Z), Positive (P), Negative (N) or Unconditional (UN) when the BCTA follows one of the instructions giving a positive, zero or negative condition code. An example will make it clearer.

```
LODA,R3     09 F0
BCTA,P      0A B0
BCTA,N      0A C0
BCTA,Z      0A D0
```

The first instruction loads R3 with the contents of address 9F0. The second instruction tells the processor to branch to address AB0 if the value loaded into R3 is positive (P). If the value loaded is not positive the processor will execute the next instruction which tells it to branch to address AC0 if the value loaded is negative (N). If the value is not negative the processor will execute the next instruction shown which tells it to branch to address AD0 if the value previously loaded was zero (Z). Clearly if the last instruction is reached the value must have been zero, so the instruction could also have been BCTA,UN OADO which means branch unconditionally, i.e. no matter what the condition code had been set to.

The values of 'con' are shown below.

Condition	'Con'
Zero (Z)	0
Positive (P)	1
Negative (N)	2
Unconditional (UN)	3

The value of 'con' is added to the instruction code as shown in the individual instructions.

After COMPARE instructions 'con' can have the conditions Equal to (EQ), Greater than (GT) or Less than (LT) when the values of 'con' are:

Condition	'Con'
Equal to (EQ)	0
Greater than (GT)	1
Less than (LT)	2

After the TEST UNDER MASK IMMEDIATE instruction (paragraph 6.5) 'con' can have the conditions 0 if all the bits tested are 1's and 1 if all the bits tested are not 1's.

5.4. Logical Functions

There are three Logical instructions available to the processor. These are AND, IOR and EOR.

AND

When two bytes are ANDed together each pair of bits in turn are combined together to give a result according to the truth table below.

Bit (7-0)	Bit (7-0)	AND result. Bit (7-0)
0	0	0
0	1	0
1	0	0
1	1	1

i.e. the corresponding bit is set to 1 only if both source bits are 1's.

Examples: 10111001 AND 01101110 = 00101000

IOR (inclusive OR)

When two bytes are IORed together each pair of bits in turn are combined together to give a result shown in the truth table below.

Bit (7-0)	Bit (7-0)	IOR result. Bit (7-0)
0	0	0
0	1	1
1	0	1
1	1	1

i.e. the corresponding bit is set to 1 if either or both source bits are 1's.

Examples: 10111001 IOR 00101110 = 10111111

EOR (Exclusive OR)

When two bytes are EORed together each pair of bits in turn are combined together to give a result according to the truth table below.

Bit (7-0)	Bit (7-0)	EOR result. Bit (7-0)
0	0	0
0	1	1
1	0	1
1	1	0

i.e. the corresponding bit is set to 1 if either of the source bits are 1's but not if both are 1's.

Examples: 10111101 EOR 11011001 = 01100100

5.5. Indirect Addressing

It is possible to use a form of addressing called indirect addressing which is not used very often but can sometimes save a lot of assembly code. The instruction takes the form LODA,Rn *HHHH and this means:- Load into Rn the contents of the address which is the contents of address HHHH and its following byte. An example will make this clearer. Suppose we have a section of program -

		Addr.	Code
LODA,R0	*0D00	0950	----
		0D00	0F
		0D01	10
		0F10	63

The processor will look at addresses 0D00 and 0D01, see that they hold the hex value 0F10 and load into R0 the value 63 from address 0F10.

If you INDEX the instruction as well, the index register is added to the second address. If our instruction above was LODA,R0 *0D00,R3+ and say R3 has the contents 2, then the processor would look at address 0D00 and add the new address 0F10 to the contents of R3 (after incrementing R3) i.e. the new address becomes 0F10 + (2 + 1) = 0F13, then load the contents of 0F13 into R0. On the next time this instruction was met it would increment R3 to 4 and use 0F14 (remember you can only use R0 when indexing).

Indirect addressing can be used on all instructions of two or more bytes where the 2nd (or 2nd and 3rd) byte is relative or absolute address. This includes branch instructions. The hexadecimal code for the instruction is formed by taking the code for the normal instruction and adding 8 to the most significant address digit i.e. the 3rd digit of the code. For example the code for LODA,R2 0D00 is 0E0D00 and for the same instruction Indirect, LODA,R2 *0D00 is 0E8D00.

5.6. Relative Addressing

Relative addressing is a form of addressing which can save memory storage in your program since relative address only requires one byte instead of two. However, it can only be used for addresses located within + or - decimal 64 (less than + or - hex 40). As the name implies a relative address is calculated from the current address: actually from the address of the next instruction following the relative address. Suppose we take as an example LOAD RELATIVE, R0 ... HH (LODR,R0 HH) = 08HH at address 0A00 and 0A01; this instruction would load R0 with the contents of an address found by adding the displacement HH to the next instruction address i.e. 0A02 + HH where HH can be +ve or -ve (see two's complement numbers) e.g. if HH is 30, R0 would be loaded from 0A32. Actually the first bit (bit 7) of HH is not part of the relative displacement, as it is used to indicate whether INDIRECT addressing is also being used (see INDIRECT addressing). Furthermore the second bit (bit 6) is used as the sign bit, so we now see why the displacement is only +64 to -63.

To give an example of a -ve displacement (without Indirect addressing), - hex 3 would give HH = 7D (-ve displacements will start with 4, 5, 6 or 7).

5.7. Subroutines

You will find that your programs will often contain blocks of code which are identical. For example you will need to read a block of data from one set of addresses and write it to another set of addresses and it would be laborious to repeat this same block of code each time you need it. You can in fact make the block of code into a Subroutine which is rather like a small completely separate program. This Subroutine can be used as often as you like simply by using an instruction which tells the processor to branch to your subroutine. When you use one of these instructions the processor does a number of things automatically. First it saves a 'Return Address' in an eight deep Return Address Stack, secondly it sets a 'Stack Pointer' which indicates the correct Return Address to be used next and thirdly it branches to your subroutine address and begins to execute the instructions there.

The Return Address that is saved is the address of the next instruction following the Branch to Subroutine instruction so when the processor finishes the subroutine and encounters a Return instruction it will return and execute that instruction next.

The Return Address Stack is eight levels deep so that you can use up to eight levels of 'nested' subroutines i.e. your program can call a subroutine which calls another subroutine which calls another subroutine up to eight levels deep and the processor will automatically keep track of all the Return Addresses for you.

There are certain rules about using subroutines so that you avoid getting into endless loops with them and it is suggested that you (1) Always use a Return instruction to get back from a subroutine and not a Branch instruction; (2) Never call a higher level subroutine from a lower one.

It is often necessary for you to transfer information into and out of subroutines. This can be accomplished by using the processor registers if only a small amount of data needs to be transferred or writing to and reading from reserved memory addresses for larger amounts of data.

Finally when you use a subroutine you should make sure that using it does not affect your main program in some incorrect manner. For example, if you are calling a subroutine from the middle of a loop which uses say R3 as a loop counter, you must make sure R3 is not altered in the subroutine. There are a number of ways of doing this; e.g. it is often possible to reserve register bank 1 for subroutines only so that by changing to bank 1 in your subroutine you automatically protect your bank 0 registers (except of course R0). Alternatively you can temporarily store your registers in memory and reload them at the end of the subroutine.

Some examples of using subroutines will be found in the Survival Game at the end of this manual.

5.8. Interrupts

An Interrupt is a signal from an external source that tells the processor that something outside needs attention. If the Interrupt Inhibit bit of the Program Status Word is 0 then interrupts are allowed and on getting one, the processor will complete the instruction it is doing and execute a Branch to Subroutine at address 0903 where you should locate your interrupt handling subroutine. In addition the processor will set the Interrupt Inhibit bit to 1 which will prevent any further interrupts until you clear it to 0. (It will be cleared automatically when you return from the interrupt subroutine with a RETE instruction).

On returning from the interrupt subroutine the processor will carry on with the next instruction following the one it was doing when interrupted. An Interrupt counts as one level of the eight levels available for subroutine nesting.

In this processor system there are interrupts available which signal when each object or its duplicate has been completed and also every 20 milli-seconds at the end of each t.v. frame (complete picture). The 20ms interrupts can also be used for timing or delay purposes if required or for repeatedly scanning the keyboard and/or joysticks at regular intervals.

An example of using the interrupt facility is given in example 3.

5.9. Instruction Timing

It is sometimes necessary to know how long each instruction takes to execute. This can occur if either you need to do something in a certain limited time or alternatively you want to use an instruction repeatedly to obtain a time delay.

The instructions are timed by 'clock cycles' of a crystal oscillator and below is a table of instruction types and the number of cycles each takes.

Instruction Type	Number Of Cycles
All BRANCH and RETURN instructions	3 *
All ABSOLUTE address instructions (except BRANCH)	4 *
All RELATIVE address instructions	3 *
All IMMEDIATE address instructions	2
All ZERO address instructions	2
RRR, RRL, NOP, LPSU, LPSL, SPSU, SPSL	2
TMI, TPSU/L, PPSU/L, CPSU/L, DAR	3
HALT	Until Reset or Interrupt

One cycle = 3.38 micro-seconds which is 3.38/1,000,000 seconds.

* For indirect addressing 2 cycles have to be added.

6. ADVANCED INSTRUCTIONS

6.1 Relative Addressing

Relative addressing, discussed in the context of branching, can also be applied to LOAD, STORE and other instructions.

LOAD RELATIVE (LODR, Rn HH): 08+n

Loads Rn (n = 0 to 3) from a memory address found by adding the displacement (bits 6 to 0 of HH) to the address of the byte following HH. The previous contents of Rn are lost.

Example: LODR, R1 08 (09 0 8). Loads R1 from the address found by adding 8 to the address of the byte following 08.

STORE RELATIVE (STRR, Rn HH): C8+n

Stores Rn (n = 0 to 3) in the address found by adding the displacement (bits 6 - 0 of HH) to the address of the byte following HH. The contents of Rn are unchanged and the contents of the memory address replaced.

Example: STRR, R0 57 (C8 57). Stores the contents of R0 in the address found by adding hex. 57 (i.e. minus hex. 29) to the address of the byte following 57.

ADD RELATIVE (ADDR, Rn HH): 88 + n

Adds the contents of Rn (n = 0 to 3) to the contents of the address found by adding the displacement (bits 6 to 0 of HH) to the address of the byte following HH and replaces the original contents of Rn with the result of the addition. The contents of the memory address are unchanged.

Example: ADDR, R3 10 (8B 10). Adds the contents of R3 to the contents of the address found by adding 10 to the address of the byte following 10 and puts the result in R3.

SUBTRACT RELATIVE (SUBR,Rn HH): A8 + n

Subtracts the contents of the address, found by adding the displacement (bits 6 to 0 of HH) to the address of the byte following HH, from Rn (n = 0 to 3) and replaces the original contents of Rn with the result of the subtraction. Subtraction is done by forming the two's complement and adding. The contents of the memory address are unchanged.

Example: SUBR,R2 61 (AA 61). Subtracts from R2 the contents of the address found by adding hex 61 (minus 1F) to the address of the byte following 61. The result is put into R2.

6.2 Logical Instructions

AND, IOR, EOR TO REGISTER ZERO (ANDZ Rn): 40 + n
(IORZ Rn): 60 + n
(EORZ Rn): 20 + n

The contents of Rn (n = 0 to 3) are combined with the contents of R0 according to the truth tables. The result replaces the contents of R0. Rn is unchanged.
N.B. The code 40 is reserved for HALT.

Example: EORZ R0 (20). R0 is EORed with itself and the result put into R0. i.e. this instruction effectively loads 00 into R0 whatever its contents are. Try this for yourself!

AND, IOR, EOR IMMEDIATE (ANDI,Rn HH): 44 + n
(IORI,Rn HH): 64 + n
(EORI,Rn HH): 24 + n

The contents of Rn (0 to 3) are combined with the byte HH according to the truth tables and the result put into Rn, replacing the original contents.

Example: IORI,R3 A7 (67 A7). The contents of R3 are IORed with A7 and the result put into R3.

AND, IOR, EOR RELATIVE	(ANDR,Rn	HH):	48 + n
	(IORR,Rn	HH):	68 + n
	(EORR,Rn	HH):	28 + n

The contents of Rn (n = 0 to 3) are combined, according to the truth tables, with the contents of the address obtained by adding the displacement (bits 6 to 0 of HH) to the address of the byte following HH. The result is put into Rn, replacing the original contents.

Example: ANDR,R1 09 (49 09). The contents of R1 are ANDed with the contents of the address found by adding 09 to the address following 09. The result is put into R1.

AND, IOR, EOR ABSOLUTE	(ANDA,Rn	HH HH):	4C + n
	(IORA,Rn	HH HH):	6C + n
	(EORA,Rn	HH HH):	2C + n

The contents of Rn (n = 0 to 3) are combined, according to the truth tables, with the contents of address HH HH. The result is put into Rn replacing the original contents.

Example: IORA,R1 IFA0 = (6D 1F A0). The contents of R1 are IORed with the contents of address 1F A0 and the result replaces the original contents of R1.

6.3 Compare

COMPARE TO REGISTER ZERO (COMZ Rn): E0 + n

The contents of Rn (n = 0 to 3) are compared with the contents of R0. The comparison will be made in either 'arithmetic' or 'logical' mode depending on the setting of the COM bit in the Program Status Word. When COM = 1 (logical mode) the values will be interpreted as 8 bit +ve binary numbers and when COM = 0 (arithmetic mode) the values will be taken to be 8 bit two's complement numbers. The result of the comparison affects the condition code as follows:

	Condition Code
R0 greater than Rn	0 1
R0 equal to Rn	0 0
R0 less than Rn	1 0

Example: COMZ R3 (E3). Compares R3 with R0 and alters the condition code accordingly.

COMPARE IMMEDIATE (COMI,Rn HH): E4 + n

The contents of Rn (n = 0 to 3) are compared with HH. The comparison mode depends on the setting of the COM bit (see COMPARE TO REGISTER ZERO). The results of the comparison affect the setting of the condition code as follows:

	Condition Code
Rn greater than HH	0 1
Rn equal to HH	0 0
Rn less than HH	1 0

Example: COMI,R1 37 = (E5 37). Compares R1 with 37 and alters the condition code accordingly.

COMPARE RELATIVE (COMR,Rn HH): E8 + n

The contents of Rn (n = 0 to 3) are compared with the contents of the address found by adding the displacement (bits 6 to 0 of HH) to the address of the byte following HH. The comparison mode depends on the setting of the COM bit (see COMPARE TO REGISTER ZERO). The results of the comparison affect the setting of the condition code as follows:

	Condition Code
Rn greater than memory byte	0 1
Rn equal to memory byte	0 0
Rn less than memory byte	1 0

Example: COMR,R2 10 (EA 10). Compares R2 with the contents of the address found by adding 10 to the address following 10 and alters the condition code accordingly.

COMPARE ABSOLUTE (COMA,Rn HH HH): EC + n

The contents of Rn (n = 0 to 3) are compared with the contents of address HH HH. The comparison mode depends on the setting of the COM bit (see COMPARE TO REGISTER ZERO). The results of the comparison affect the condition code as follows:

	Condition Code
Rn greater than memory byte	0 1
Rn equal to memory byte	0 0
Rn less than memory byte	1 0

Example: COMA,R3 1FA0 (EF 1F A0). Compares Rn with the contents of address 1F A0 and affects the condition code accordingly.

6.4 Conditional Branching

BRANCH ON CONDITION TRUE, ABSOLUTE
(BCTA,con HHHH): 1C + con

This instruction causes the processor to branch to the address HH HH if the condition specified ('con' = 0 to 3) is the same as the Condition Code held in the Program Status word (set on execution of a previous instruction). If 'con' is not the same as the Condition Code the next instruction (following the bytes HH HH) is executed.

Example: LODZ R3 (Condition Code is set)
BCTA,P 0A 00 = (1D 0A 00)

If the value of R3 loaded into R0 was positive (CC= 1), the processor will branch to address 0A 00 and execute the instruction there. If the value loaded was zero or negative the processor will execute the next instruction (following the bytes HH HH).

BRANCH ON CONDITION TRUE, RELATIVE

(BCTR,con HH): 18 + con.

This instruction behaves exactly the same as the previous instruction (BCTA,con) except that the branch address is found by adding the displacement (bits 6 to 0 of HH) to the address of the byte following HH.

Example: BCTR,UN 20 (1B 20). The processor will branch unconditionally (no matter what the Condition Code is set to) to the address found by adding 20 to the address of the byte following HH.

BRANCH ON CONDITION FALSE, ABSOLUTE

(BCFA, con HH HH): 9C + con.

This instruction causes the processor to branch to the address HH HH if the condition specified ('con' = 0 to 3) is not the same as the Condition Code setting in the Program Status Word (set on execution of a previous instruction). If 'con' is the same as the Condition Code the processor executes the next instruction (following the bytes HH HH).

```
Example:   COMI,R0    30   (Condition Code is set)
           BCFA,LT    0B 20     (9E 0B 20)
```

The processor will branch to address B20 if R0 is not less than hex. 30. If R0 is less than 30 the next instruction will be executed.

BRANCH ON CONDITION FALSE, RELATIVE

(BCFR,con HH): 98 + con.

This instruction behaves exactly the same as the previous instruction except that the branch address is found by adding the displacement (bits 6 to 0 of HH) to the address of the byte following HH.

Example: BCFR,UN 30 (9B 30). The processor will unconditionally branch to the address found by adding 30 to the address of the byte following HH.

BRANCH ON REGISTER NON ZERO RELATIVE

(BRNR,Rn HH): 58 + n

The contents of Rn (n = 0 to 3) are tested for a non-zero value. If the value is non-zero the processor will branch to the address obtained by adding the displacement (bits 6 to 0 of HH) to the address of the byte following HH. If the register value is zero the processor will execute the next instruction following HH.

Example: BRNR,R3 10 (5B 10). This causes the processor to branch to the address found by adding hex. 10 to the address following 10 if the contents of R3 are not zero. If the contents of R3 are zero the next instruction will be executed at the address following 10.

BRANCH TO SUBROUTINE ON NON-ZERO REGISTER, ABSOLUTE

(BSNA,Rn HH HH): 7C + n

The contents of Rn (n = 0 to 3) are tested for a non-zero value. If the value is non-zero the processor will branch to the subroutine address HH HH and execute that instruction having first incremented the Stack Pointer by one and put the address of the next instruction (at the address following the bytes HH HH) in the Return Address Stack.

If Rn contains a zero value the next instruction will be executed.

Example: BSNA,R1 0A70 (7D 0A 70). The processor will branch to a subroutine at address A70 if the value of R1 is non-zero. If the value of R1 is zero, the next instruction will be executed.

BRANCH TO SUBROUTINE ON NON-ZERO REGISTER, RELATIVE

(BSNR,Rn HH): 78 + n

This instruction behaves exactly the same as the previous instruction (BSNA,Rn except that the subroutine address is found by adding the displacement (bits 6 to 0 of HH) to the address of the next instruction (following HH).

Example: BSNR,R2 30 (7A 30). If the value of R2 is zero, the next instruction is executed. If the value of R2 is non-zero, the instruction executed is at the address found by adding the displacement to the next address. (After storing the Return address and incrementing the Stack Pointer).

BRANCH ON INCREMENTING REGISTER, ABSOLUTE

(BIRA,Rn HH HH): DC + n

The contents of Rn (n = 0 to 3) are incremented by one and if the new value in Rn is non-zero, the processor branches to the address HH HH and executes the instruction there. If the new value of Rn is zero the processor executes the next instruction (following the bytes HH HH).

Example: BIRA,R3 1000 (DF 10 00). R3 is incremented by 1 and if the new value is non-zero the processor branches to address 1000. If R3 is zero the next instruction is executed.

BRANCH ON INCREMENTING REGISTER, RELATIVE

(BIRR,Rn HH): D8 + n

This instruction behaves exactly the same as the previous instruction (BIRA,Rn) except that the branch address is found by adding the displacement (bits 6 to 0 of HH) to the address of the next instruction (following HH).

Example: BIRR,R1 10 (D9 10). The value of R1 is incremented by one and if the new value is non-zero the processor branches to the address found by adding hex. 10 to the address of the byte following HH. If the new value is zero the processor executes the next instruction at the address of the byte following HH.

BRANCH ON DECREMENTING REGISTER, ABSOLUTE

(BDRA,Rn HH HH): FC + n

The contents of Rn (n = 0 to 3) are decremented by one. If the new value of Rn is non-zero the processor branches to address HH HH and executes the instruction there. If the new value of Rn is zero, the processor executes the next instruction (following the bytes HH HH).

Example: BDRA,R2 12 20 (FE 12 20). R2 is decremented by one and if the new value is non-zero the processor branches to address 12 20. If R2 becomes zero the next instruction is executed (following the bytes 12 20).

BRANCH ON DECREMENTING REGISTER, RELATIVE

(BDRR,Rn HH): F8 + n

This instruction behaves exactly the same as the previous one (BDRA, Rn) except that the branch address is found by adding the displacement (bits 6 to 0 of HH) to the address of the byte following HH.

Example: BDRR,R3 7E (FB 7E). This instruction is often used to introduce a time delay. It causes the processor to branch to its own address (displacement of -2) and execute itself for a number of times equal to the value of R3 and then when R3 becomes zero to pass on to the next instruction. If R3 was initially zero for example, it would decrement to FF (-1) and continue to loop for 256 times then the processor would continue with the next instruction thus giving a delay of 256 x 3 x 3.38 micro-sec. or approx. 2.6 milli-sec.

BRANCH TO SUBROUTINE ON CONDITION TRUE, ABSOLUTE

(BSTA,con HH HH): 3C + con

If the condition (con = 0 to 3) matches the Condition Code in the Program Status Word (set by a previous instruction), the processor will perform a subroutine branch to address HH HH and execute the instruction there. The Stack Pointer will be incremented by one and the address of the instruction following the bytes HH HH will be placed in the Return Address Stack.

If the condition code does not match, the processor will execute the next instruction following the bytes HH HH.

Example: COMI,R3 16 (Condition Code will be set)
BSTA,EQ 0D30 = (3C 0D 30)

The processor will branch to the subroutine of address D30 if R3 = hex. 16. If R3 is not equal to 16 the processor will execute the next instruction.

BRANCH TO SUBROUTINE ON CONDITION TRUE, RELATIVE

(BSTR,con HH): 38 + con

This instruction behaves exactly the same as the previous instruction except that the branch address is found by adding the displacement (bits 6 to 0 of HH) to the address of the byte following HH.

Example: LODA,R1 1F90 (Condition Code will be set)
BSTR,N 0C = (3A 0C)

If R1 has been loaded with a negative number (bit 7 = 1), the processor will branch to a subroutine located at 0C plus the address of the byte following 0C. If R1 is not negative, the next instruction (following the byte 0C) will be executed.

BRANCH TO SUBROUTINE ON CONDITION FALSE, ABSOLUTE

(BSFA,con HH HH): BC + con

If the condition ('con' = 0 to 3) does not match the Condition Code in the Program Status word, the processor will branch to a subroutine located at address HH HH. The Stack Pointer will be incremented by one and the address of the next instruction (following the bytes HH HH) will be placed in the Return Address Stack.

If 'con' does match the Condition Code, the processor will execute the next instruction following the bytes HH HH.

Example: COMA,R3 1F3F (Condition Code will be set)
BSFA,EQ 0E00 (BC 0E 00).

If the contents of R3 are not equal to the contents of address 1 F 3F the processor will branch to subroutine at address E00. If R3 is equal to the contents of 1F3F the next instruction (following 0E 00) will be executed.

BRANCH TO SUBROUTINE ON CONDITION FALSE, RELATIVE

(BSFR,con HH): B8 + con

This instruction behaves exactly the same as the previous instruction except that the branch address is found by adding the displacement (bits 6 to 0 of HH) to the address of the byte following HH.

Example: BSFR,UN 50 (BB 50).

The processor will unconditionally branch to subroutine at address found by adding 50 to the address of the next instruction (following 50).

RETURN FROM SUBROUTINE, CONDITIONAL

(RETC,con): 14 + con

This instruction is used to conditionally return the processor to the program which last issued a subroutine branch instruction.

If the condition ('con' = 0 to 3) matches the Condition Code in the Program Status Word, the top address in the Return Address Stack is taken as the address of the next instruction and the Stack Pointer is decremented by one.

If the condition does not match the return is not effected and the processor will execute the next instruction following the RETC instruction.

Example: RETC,UN (17). The processor will return unconditionally irrespective of the Condition Code setting.

RETURN FROM SUBROUTINE AND ENABLE INTERRUPT, CONDITIONAL

(RETE,con): 34 + con

This instruction is mainly intended to be used by an interrupt handling routine because receipt of an interrupt causes a subroutine branch and additionally sets the Interrupt Inhibit bit to 1 (in the Program Status Word) to prevent further interrupts. Using this instruction conditionally effects a return to the program which last issued a subroutine branch (or the next instruction from where an Interrupt occurred) and additionally clears the Interrupt Inhibit bit thus immediately allowing further interrupts.

If 'con' matches the Condition Code return is effected, the next instruction executed will be at the address at the top of the Return Address Stack and the Stack Pointer decremented by one.

If 'con' does not match the Condition Code the next instruction executed will be the one following the RETE instruction.

```
Example:  LODA,R1   1FE8   (Condition Code is set)
          RETE,P           (35)
```

Return is effected only if the new contents of R1 are positive (bit 7 = 0). If R1 is not positive, the next instruction executed is the one following the byte 35.

6.5 Test Under Mask Immediate

(TMI,Rn HH): F4 + n

If the eight bits in a byte are used separately rather than as a binary number, it is often a requirement to test one or more bits separately. A mask is used, which is a byte with a 1 in the position of each bit in which we are interested. Thus H'81' = B'1000 0001' is a mask to test the first (bit 0) and eighth (bit 7) bits of a byte.

The condition codes are set as follows:

All bits masked are ones 00
Not all bits masked are ones 10

```
Example:   TMI,R3   81    (F7 81)
           BCTA,N1  HHHH (1E HH HH)
```

will branch to HHHH if either bit 0 or bit 7 of R3 is not a 1.

Only immediate addressing is applicable to this instruction; the mask cannot be accessed in RAM or another register.

6.6 Rotate Register

(RRR,Rn): 50 + n (RRL,Rn): D0 + n

If the contents of a register are rotated (or shifted) one place to the left the number is doubled, or to the right, halved; this can be used for multiplication and division. If it is desired to operate on individual bits of a byte it is convenient to move them to a predetermined position in the register - say to the left hand end; when a byte is shifted either way the end bit 'drops off' one end, and the bit at the other end is undefined. In a ROTATE instruction it is therefore arranged that the end bit carries round to enter the register at the other end, so that eight rotations restore the byte to its original value. If this is not wanted these bits can be removed by a logical mask, e.g.:

```
RRR,R0                  Divide by 4
RRR,R0
ANDI,R0  3F             Mask off bits 6 and 7
```

If the WC bit in the Program Status word is set the end bit is moved to the CARRY bit and the old CARRY bit to the other end of the register, so it would take 9 rotations to restore the byte to its original value. Also, Register bit 6 flows into the IDC if (WC) = 1.

Example: Multiply R1 by 9.

LODZ R1	01	R0 = R1
RRL, R0	50	R0 = 2 x R1
RRL, R0	50	R0 = 4 x R1
RRL, R0	50	R0 = 8 x R1
ADDZ, R1	81	R0 = 9 x R1
STRZ R1	C1	R1 = R0

6.7 Decimal Adjust

(DAR,Rn): 94 + n

It is sometimes necessary to use decimal, rather than hexadecimal, arithmetic, such as when calculating scores, mathematical games etc. - you can even use your hobby module as a programmable calculator.

As the decimal digits 0 to 9 correspond to binary 0000 to 1001, only four bits (one nibble) are needed to hold one digit, so two digits can be held in one 8 bit byte. A decimal number such as D'35' would appear in the register as H'35'. If arithmetic functions are carried out on it however, it will be treated as a binary number (which we look at as hexadecimal) and answers such as 48 + 35 = 7D will appear. However the DAR instruction will turn this into two decimal digits. It does this by looking at the carry from the first nibble into the second (another Program Status Word bit, called Inter Digit Carry), and the CARRY bit, when 6 is added to each nibble (i.e. hex. 66 to the byte). This must be done by the program; the DAR will then automatically sort out the correct answer. The procedure for adding two numbers is therefore: (1) load one byte into a register, (2) add hex. 66 to it, (3) add the second number to it, and (4) DAR.

Take, as an example, adding a number - say decimal 15 - to the score digits.

```
        LODI,R0 3       0900 04 03      Set score to 4 figures
        STRA,R0 1FC3    0902 CC IF C3   and bottom of screen
        CPSL FF         0905 75 FF
        PPSL WC         0907 77 08      Prepare to add with carry
Count   LODA,R1 0B00    0909 0D 0D 00   Load previous lower byte
        LODA,R2 0B01    090C 0E 0B 01   Load previous upper byte
        ADDI,R1 66      090F 85 66      Prepare for DAR
        ADDI,R2 66      0911 86 66
        CPSL 1          0913 75 01      Clear carry bit
        ADDI,R2 15      0915 86 15      Add 15
        DAR,R2          0917 96         Decimal adjust lower
        ADDI,R1 0       0918 85 00      Add carry to upper byte
        DAR,R1          091A 95         Decimal adjust upper
        STRA,R1 0B00    091B CD 0B 00   Store answers back
        STRA,R2 0B01    091E CE 0B 01
        STRA,R1 1FC8    0921 CD 1F C8   Store answer in score
        STRA,R2 1FC9    0924 CE 1F C9
Wait    LODA,R0 1FCB    0927 0C 1F CB   Wait for VRLE
        TMI, R0 40      092A F4 40
        BCTR,N1 Wait    092C 1A 79
        BCTA,UN Count   092E 1F 09 09
```

The instruction in location 927 to 92E are added to test the program - after a 20m-sec wait the program repeats, causing continuous adding.

If the subtract command is used with DAR it is not necessary to add 66 first so CARRY and IDC are set automatically.

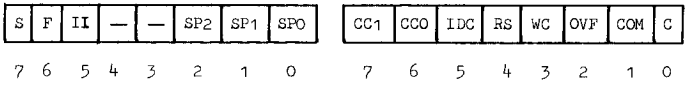
```
e.g. LODA,R1  ADDR1  0D HH HH
     SUBI,R1  6      45 06
     DAR,R1           96
     etc.
```

However if zero is subtracted from a register the DAR instruction gives the wrong answer, and this must be avoided.

```
e.g.   LODA,R0    ADDR1     00  0A  A8
       LODA,R1    ADDR2     0D  0A  A9
       BCTR,Z     NOSUB     18  01
       SUBZ  R1             A1
NOSUB  STRA,R0    ADDR1     CC  0A  A8
```

6.8 Program Status Word

This has already been mentioned in connection with other instructions. It consists of two bytes (the Upper and Lower bytes) being a collection of flags for a variety of purposes:



PROGRAM STATUS UPPER (PSU) PROGRAM STATUS LOWER (PSL)

S = Sense, vertical synchronisation signal (VRLE), indicates the flyback of the screen sweep.

F = Flag, used to distinguish axes of joysticks (paragraph 3.7)

II = Interrupt Inhibit (paragraph 5.8)

SP0 to SP2 = stack pointer, used to indicate level of subroutine nesting. The microprocessor updates these automatically when entering or returning from a subroutine or when handling an interrupt.

CC0, CC1 = Condition Codes (paragraph 5.3)

IDC = Inter-digit carry (paragraph 6.7)

RS = Register Bank Select (paragraph 1.3)

WC = With Carry (paragraph 5.1)

OVF = Overflow (paragraph 5.2)

COM = Compare Logically or Arithmetically (paragraph 6.3)

C = Carry (paragraph 5.1)

The Program Status Word can be examined and altered by the following commands:-

PRESET MASKED BITS

(PPSL mask): 77 MM and (PPSU mask): 76 MM

Sets the masked bits in MM to a 1

Example: PPSL 18 sets RS and WC (Reg. bank 1 and with-carry-arithmetic) as hex. 18 = 0001 1000

CLEAR MASKED BITS

(CPSL mask): 75 MM and (CPSU mask): 74 MM

Sets the masked bits in MM to a 0

Example: CPSL FF sets all bits to zero as hex. FF = 1111 1111

TEST MASKED BITS

(TPSL mask): 85 MM and (TPSU mask): 84 MM

Sets Condition Code to 00 if masked bits all 1, otherwise 10.

```
Example:  TPSL  04
          BCTA,A1   ERROR     tests for overflow after arithmetic operation.
```

Of the upper byte bits only the Interrupt Inhibit and flag bits will be commonly used, and only these are listed in the Instruction Code Summary, Appendix 1.

7. EXAMPLES OF PROGRAMMING

7.1. Example 1 - Setting Up Objects and Background

First, we set up the Program Status Word and interrupt disable:

```
START | CPSL ALL      | 900 | 75 FF    | All bits = 0
      | PPSU II       | 902 | 76 20    | disable interrupt
```

Write an object into the PVI RAM for object 1 (using the object in figure 2).

```
      | LODI,R1 0E          | 904 | 05 0E          | Prepare to write H'0E' bytes
LOOP1 | LODA,R0 DATA1,R1,-  | 906 | 0D 49 12       | Take a byte from the data
      |                     |     |                | table below
      | STRA,R0,1F00,R1     | 909 | CD 7F 00       | Store data into object RAM
      |                     |     |                | (1F10 etc)
      | BRNA,R1 LOOP1       | 90C | 5D 09 06       | Go back for next data byte
      | BCTA,UN COLOR       | 90F | 1F 09 20       | Jump over the data table
DATA1 | DATA 34,87,FF,FC    | 912 | 84 87 FF FC    |
      | DATA FC,C6,A5,0     | 916 | FC C6 A5 00    | DATA table for object and
      |                     |     |                | duplicate
      | DATA 0,0,40,4B      | 91A | 00 00 40 4B    |
      | DATA 5,10           | 91E | 05 10          |
```

Having set an object, by a read/write data loop, now write its colour and size. For only two bytes a loop is not worth while:

```
COLOR | LODI,R0 18          | 920 | 04 18          |
      | STRA,R0 1FC1        | 922 | CC 1FC1        | Write 18 into object
      |                     |     |                | colour 1 & 2
      | LODI,R0 01          | 925 | 04 01          |
      | STRA,R0 1FC0        | 927 | CC 1FC0        | Write 01 into object size
```

If we now write 40 (HALT) into the next location (92A) and run the program from 900 we should get red dogs on the screen.

To add a background, draw out a block like figure 3, decide which sections you want to fill in, and work out the contents of locations 1F80 to 1FA7. Then decide on the extensions of each strip of section (including the overrides to set full width) and work out the contents of location 1FA8 to 1FAC. These can be entered by a read/write loop similar to the one above. Before entering this, change locations 91D and 91F to 'FF' - we do not want the duplicates anymore; also move Hamish down by changing 91E to '90' and 91C to '10'.

```
        LODI,R1 2D              92A  05 2D          Prepare to move 2D bytes
LOOP2   LODA,R0 DAT2,R1,-       92C  0D 49 38       Take byte from table
        STRA,R0 1F80,R1         92F  CD 7F 80       Store it in PVI
        BRNA,R1 LOOP2           932  5D 09 2C       Go back for next byte
        BCTA,UN BKGON           935  1F 09 64       Jump over data table
DAT2    DATA 0,18,0,1C          938  00 18 00 1C    Data table
        DATA 0,14,0,3C          93C  00 14 00 3C
        DATA 0,2C,0,18          940  00 2C 00 18
        DATA 0,18,0,8           944  00 18 00 08
        DATA 0,E8,7,88          948  00 E8 07 88
        DATA 7,10,78,10         94C  07 10 78 10
        DATA F0,E0,0,E0         950  F0 E0 00 E0
        DATA 0,0,FF,80          954  00 00 FF 80
        DATA 0,0,0,0            958  00 00 00 00
        DATA 0,0,0,0            95C  00 00 00 00
        DATA D5,46,49,49        960  D5 46 49 49    Extension data
BKGON   LODI,R0 29              964  04 29          Turn on background
        STRA,R0 1FC6            966  CC 1F C6       and colour
        HALT                    969  40             and stop
```

7.2. Example 2 - Joystick Control, Moving Objects, Wait Loops And Bank 1

We shall continue with the program of example 1 to demonstrate joysticks. To control position with the joystick simply requires reading the joystick and writing this number (possibly modified) into either the vertical or horizontal object position byte. A little more advanced is the idea of controlling speed of movement. If the position is changed by the same amount in equal time intervals the object moves at a constant speed. Equal time intervals can be obtained by looking at bit 6 of location 1FCB, which is set high when the picture scan finishes - every 20 milliseconds - and is reset when it is read, to ensure that it cannot be read twice in the same interval.

Continuing our program from example 1, and replacing the HALT instruction:

```
JOYST   PPSU  FLAG          969  76 40      set to forwards joystick
SYNC1   LODA,R0 1FCB        96B  0C 1F CB   look at sync pulse
        TMI,R0 40           96E  F4 40     test bit 6 (VRLE)
        BCTR,N1 SYNC1       970  1A 79     try again if 0
        LODA,R2 1FCC        972  0E 1F CC   load forward left joystick
        CPSU FLAG           975  74 40     set to lat. joystick
SYNC2   LODA,R0 1FCB        977  0C 1F CB   wait for sync pulse
        TMI,R0 40           97A  F4 40
        BCTR,N1 SYNC2       97C  1A 79
        LODA,R1 1FCC        97E  0D 1F CC   load lat. left joystick
        RRR,R2              981  52
        RRR,R2              982  52         scale number
        RRR,R2              983  52         (divide by 16)
        RRR,R2              984  52
        ANDI,R2 0F          985  46 0F      clear higher nibble
        LODA,R0 1F0C        987  0C 1F 0C   load present vert. position
        SUBZ R2             98A  A2         subtract joystick
        STRA,R0 1F0C        98B  CC 1F 0C   store new vert. position
        RRR,R1              98E  51
        RRR,R1              98F  51         scale number
```

```
        RRR,R1          990  51          (divide by 16)
        RRR,R1          991  51
        ANDI,R1 OF      992  45 OF       clear higher nibble
        ADDA,R1 1FOA    994  8D 1F OA    add to present hor. position
        STRA,R1 1FOA    997  CD 1F OA    store new hor. position
```

If we now branched unconditionally back to location 969 we should set up a loop which would monitor the left joystick and move Hamish at a speed dependent on the joystick position - try it if you want. However, there are two more things to do - make his legs move as he walks, and make sure he stays within the limits of the path.

To move his legs, re-write the appropriate location in the object block of memory if his horizontal position has a '1' in the third bit (i.e. change legs every fourth horizontal position).

```
        LODI,R2 86      99A  06 86       first leg pattern
        LODI,R3 85      99C  07 85
        TMI,R1 4        99E  F5 04       every fourth location
        BCFA,A1 CLEGS   9A0  9C 09 A7    don't change
        LODI,R2 4C      9A3  06 4C       change legs to
        LODI,R3 34      9A5  07 34       second leg pattern
CLEGS   STRA,R2 1F05    9A7  CE 1F 05    update PVI
        STRA,R3 1F06    9AA  CF 1F 06
        LODA,R0 1FCA    9AD  0C 1F CA    test for collision
        TMI,R0 80       9B0  F4 80       with background
        BCFA,A1 JOYST   9B2  9C 09 69    loop back if not
        COMI,R1 70      9B5  E5 70       hit tree?
        BCFA,GT START   9B7  9D 09 00    no - start again
        HALT            9BA  40          stop at tree
```

Now we have reached the winning post we can look at two more techniques, sound production and wait loops. When we want to pause for a set time it is often desirable to keep the register contents. This can be done by using the second register bank to time the waiting period. This is not necessary here but the technique will be shown. To count time we again use the 20 millisecond pulse in location 1FCB, bit 6 (VRLE).

```
WIN     LODI,R0 4          9BA  04 04
        STRA,R0 1E80       9BC  CC 1E 80   turn on sound
        LODI,R0 0B         9BF  04 0B
FANFR   CPSL  R5           9C1  75 10      return to bank 0 inside loop
        STRA,R0 1FC7       9C3  CC 1F C7   write sound pitch
        PPSL R5            9C6  77 10      second register bank
LOOP4   LODI,R1  06        9C8  05 06
LOOP3   LODA,R2 1FCB       9CA  0E 1F CB   wait 10μsec
        TMI,R2 40          9CD  F6 40
        BCTR,N1 LOOP3      9CF  1A 79
        BDRR,R1 LOOP4      9D1  F9 77      H'32' times = 1/3 sec.
        BDRR,R0 FANFR      9D3  F8 6C      change note and jump back
        EORZ,R0            9D5  20         clear R0
        STRA,R0 1E80       9D6  CC 1E 80   turn off sound
        HALT               9D9  40         and stop
```

7.3. Example 3 - Multiple Objects

This example will show a simple game of knocking out objects by collisions.

An interrupt is generated when (a) the PVI has completed writing any object and (b) when the picture is complete. The interrupt signal causes the program to execute a subroutine at location 903. As programs always start at 900, start by branching around the subroutine to the main program.

```
START   BCTA,UN MAIN        900   1F 0A 00
```

In the interrupt routine was the interrupt an object completion or VRLE?

```
        LODA,R3 1FCB        903   0F 1F CB     Check VRLE
        TMI,R3 40           906   F7 40
        BCTA,A1 RESET       908   1C 0A 3C
        LODA,R0 1FCA        90B   0C 1F CA     Object 3 complete?
        ANDI,R0 02          90E   44 02
        RETC,Z              910   14
```

If the interrupt was caused by completion of an object other than object 3 the program now returns from the subroutine, as we are not interested in this situation.

The interrupt subroutine is the only time at which individual 'copies' of the object can be studied, so look for collisions now. The collisions byte was reset when we accessed it for testing VRLE, but the information is still in R3.

```
        ANDI,R3 11          911   47 11       Select the appropriate
                                                collision bits
        LODZ,R3             913   03          (To set the condition
                                                codes)
        BCTR,Z NOCOL        914   18 1E       Jump round if no collision
        LODI,R0 FF          916   04 FF
        LODI,R1 03          918   05 0B       Eliminate current copy by
                                                setting hor. coord. to FF
        STRA,R0 *0F00,R1    91A   CD EF 00
```

Now add one to the score of the player achieving the collision. We cannot access the scores in the PVI memory as this would reset them, so we keep a copy in scratchpad (location 1F04 and 1F05).

```
        LODZ,R3             91D   03
        ANDI,R3 10          91E   47 10       Extract collision of
                                                objects 1 and 3
        ADDA,R3 0F04        920   8F 0F 04    (May be 0 or 1)
        STRA,R3 0F04        923   CF 0F 04    Add to score 1 in scratchpad
        STRA,R3 1FC8        926   CF 1F C8    Write to PVI
        ANDI,R0 01          929   44 0F       Repeat for second score
        ADDA,R0 0F05        92B   8C 0F 05
        STRA,R0 0F05        92E   CC 0F 05
        STRA,R0 1FC9        931   CC 1F C9
```

As the object copy has been transmitted to the screen we can now change the PVI object to the next pattern by incrementing by H'10' the number in scratchpad location 0F00/0F01 and using it as an indirect address to the next H'10' bytes of date for the next object.

```
NOCOL   LODA,R0 0F01        934   0C 0F 01
        ADDI,R0 10          937   84 10
        STRA,R0 0F01        939   CC 0F 01
```

* See Paragraph 5-5 Indirect Addressing .

To illustrate the type of function that can be carried out during the interrupt we will change the screen colour and object colours after each object. Again we must keep copies in scratchpad (locations 0F02 and 0F03).

```
        LODA,R0 0F02        93C  0C 0F 02    Screen colour
        SUBI,R0 11          93F  A4 11
        STRA,R0 0F02        941  CC 0F 02
        STRA,R0 1FC6        944  CC 1F C6    Copy to PVI
        LODA,R0 0F03        947  0C 0F 03    Object 3 colour
        ADDI,R0 08          94A  84 08
        STRA,R0 0F03        94C  CC 0F 03
        STRA,R0 1FC2        94F  CC 1F C2    Copy to PVI
WOBJ3   LODI,R1 0E          952  05 0E       Write new object into
                                             PVI copy
LOOP1   LODA,R0 *0F00,R1,-  954  0D CF 00
        STRA,R0 1F20,R1     957  CD 7F 20
        BRNR,R1 LOOP1       95A  59 78
        RETC,UN             95C  17          Return to main program
```

We shall leave some space after this subroutine to add any afterthoughts.

The main program must set up the initial condition.

```
MAIN    CPSL ALL            A00  75 FF
        PPSU II             A02  76 20       Inhibit interrupt
        LODI,R1 90          A04  05 90       Enter hor. coords of copies
LOOP2   SUBI,R1 10          A06  A5 10       To permit repeat of game
        LODA,R0 0B0E,R1     A08  0D 6B 0E
        STRA,R0 0B0B,R1     A0B  CD 6B 0B
        BRNR,R1 LOOP2       A0E  59 76
        LODI,R0 8           A10  04 08       Switch on screen colour
        STRA,R0 0F02        A12  CC 0F 02
```

* See Paragraph 5-5 Indirect Addressing .

```
        LODI,RO FA          A15  04 FA      Prepare left digit in score 1
        STRA,RO 0FO4        A17  CC 0F 04
        LODI,RO AF          A1A  04 AF      Prepare right digit in score 2
        STRA,RO 0FO5        A1C  CC 0F 05
        LODI,RO 0           A1F  04 00
        STRA,RO 1FC0        A21  CC 1F C0    Set smallest size,
                                              all objects
        STRA,RO 1FC3        A24  CC 1F C3    Set score digits separate,
                                              top of screen
        STRA,RO 1FC1        A27  CC 1F C1    Set objects 1 and 2 white
        LODI,RO 0B          A2A  04 0B       Set indirect address
        STRA,RO 0FO0        A2C  CC 0F 00    First byte to 0B
        LODI,R1 0E          A2F  05 0E       Write objects 1 and 4
WOB14   LODA,RO 0C00,R1,-   A31  0D 4C 00    To the PVI
        STRA,RO 1F00,R1     A34  CD 7F 00
        STRA,RO 1F40,R1     A37  CD 7F 40
        BRNR,R1 WOB14       A3A  59 75
```

The remainder of the setting-up needs to be done again at the end of each T.V. frame, so we shall incorporate it into the RESET program block.

```
RESET   LODI,RO 0F          A3C  04 0F      Object 4 black, and reset
        STRA,RO 0FO3        A3E  CC 0F 03    object 3 to yellow
        STRA,RO 1FC2        A41  CC 1F C2    Write colours to PVI
        LODI,RO 00          A44  04 00       Reset lower byte of indirect
        STRA,RO 0FO1        A46  CC 0F 01    Address to 0
        LODI,RO F7          A49  04 F7       Reset screen colour
        STRA,RO 1FC6        A4B  CC 1F C6    To black in scratchpad
        STRA,RO 0FO2        A4E  CC 0F 02    And write to PVI
```

We now need to write the first object to the screen, but this program block is available in the interrupt subroutine so use it again.

```
        BSTA,UN WOBJ3       A51  3F 09 52
```

With the exception of the first cycle this part of the program will be seen only at the end of the T.V. frame (as a branch to RESET) so it provides the correct timing for looking at the joysticks.

```
        LODA,RO 0F06        A54  0C 0F 06   Look at lateral/forward
                                            marker
        BCTR,P FRWRD        A57  19 15      Jump if forward marker
        LODA,RO 1FCC        A59  0C 1F CC   Joystick 1 lateral
        STRA,RO 1F0A        A5C  CC 1F 0A   Into object 1 hor. coord.
        LODA,RO 1FCD        A5F  0C 1F CD   Joystick 2 lateral
        STRA,RO 1F4A        A62  CC 1F 4A   Into object 4 hor. coord.
        PPSU FLAG           A65  76 40      Set counter to forward
                                            joystick
        LODI,RO 1           A67  04 01
        STRA,RO 0F06        A69  CC 0F 06   Set marker to forward
        BCTR,UN ENRES       A6C  1B 13      Miss out forward joystick
                                            routine
FRWRD   LODA,RO 1FCC        A6E  0C 1F CC   Forward joystick 1
        STRA,RO 1F0C        A71  CC 1F 0C   Into vert coord, object 1
        LODA,RO 1FCD        A74  0C 1F CD   Forward joystick 2
        STRA,RO 1F4C        A77  CC 1F 4C   Into vert coord, object 4
        CPSU FLAG           A7A  74 40      Set counter to lateral
                                            joystick
        LODI,RO 0           A7C  04 00      Clear marker in scratchpad
        STRA,RO 0F06        A7E  CC 0F 06
```

We are now at the end of the RESET subroutine, but cannot RETC, as we can also enter it directly from the main program, and this would cause chaos. The way out of the dilemma is to reset the stack pointer to zero, which causes the system to forget if it was in a subroutine. We must also now permit interrupts so that the system can start to write the object copies.

```
CPSU II+SP          A81   74 27
BCTR,UN ENRES       A83   1B 7C
```

This last line causes a loop which waits for the interrupt, then enables the next interrupt after each is completed.

We now need the data blocks for the object copies:-

```
0B00 00, FF, 00, 00, 00, 00, 00, 00, 00, 00, 5C, FF, 1E, 10, FF, 00
0B10 99, 99, 99, 99, 99, 99, 99, 99, 99, 99, FF, 00, FF, 10, 18, 00
0B20 AA, AA, AA, AA, AA, AA, AA, AA, AA, AA, FF, 00, FF, 10, 45, 00
0B30 0F, F0, 0F, F0, 0F, F0, 0F, F0, 0F, F0, FF, 00, FF, 10, 98, 00
0B40 81, C3, E7, 7E, 3C, 3C, 7E, E7, C3, 81, FF, 00, FF, 10, 30, 00
0B50 81, 81, 81, 81, FF, FF, 81, 81, 81, 81, FF, 00, FF, 10, 70, 00
0B60 FF, FF, C3, C3, C3, C3, C3, C3, FF, FF, FF, 00, FF, 10, A5, 00
0B70 18, 3C, 66, C3, 81, 81, C3, 66, 3C, 18, FF, 00, FF, 10, 20, 00
0B80 FF, 7E, 3C, 18, 18, 18, 18, 3C, 7E, FF, FF, 00, FF, 10, 80, 00
0B90 81, 42, 24, 99, 7E, 7E, 99, 24, 42, 81, FF, 00, FF, 10, 40, 00
```

And finally, the data block for the movable objects (1 and 4):

```
0C00 3C, 7E, 7E, FF, FF, FF, FF, 7E, 7E, 3C, 58, FF, 60, FF
```

In these data blocks the first ten bytes define the object and the following four the object and copy position. In this program, of the four position bytes, the first and third are irrelevant as all objects (except the first) are "copies", the second is irrelevant as it is overwritten in the program (locations A04 to A0E) by the byte following the four position bytes. This is done so that if the game is played more than once the objects that have been eliminated in the first game will automatically be restored; otherwise the program would have to be re-loaded or the data corrected by hand.

The last position byte, the copy separation, is important as it defines the time available to the microprocessor to carry out the interrupt routine, which it must finish in order to re-define the objects before they are copied. The best way to find the minimum value (i.e. the largest number of object copies) is by trial.

Finally the 15th data byte of each block is a zero, to pad the data block to a round figure of H'10'. It is available for further extensions of the game - for example object 2 could be treated the same way as object 3 and the number of objects doubled.

7.4. **Example 4** - The Survival Game

The program puts a pattern of small blocks on the T.V. screen in a 16 by 20 matrix of elements.

The processor takes each element in turn and examines the surrounding 8 elements (the matrix is assumed to be surrounded by blank elements). If there are 2 blocks present in the 8 elements, the element being tested remains the same. If there are 3 blocks present in the 8 elements, the tested element is replaced by a block for the next display. If any other number of blocks surround the tested element it is replaced by a blank.

When all the elements have been tested, the T.V. display is changed to the new matrix.

The game is to find a starting pattern which survives for the longest time i.e. not vanish, fill up with blocks or oscillate between different patterns.

The program has a built-in starting pattern to give you a target to beat and you can change it to your own pattern as follows:-

1. Reset and put a Breakpoint at address 929.

2. Press PC and put in 900; press '+'.

3. Enter any pattern you wish by altering the contents of addresses F90 to FB7.

4. Press PC and '+' and run the program from 929 with your matrix.

The built-in matrix is shown below.

Address	Bit No. 7 6 5 4 3 2 1 0	Bit No. 7 6 5 4 3 2 1 0	Address
F90	1	1	F9A
91	1	1	9B
92	1	1	9C
93	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	9D
94	1	1	9E
95	1	1	9F
96	1	1	A0
97	1	1	A1
98	1	1	A2
99	1	1	A3
FA4	1	1	FAE
A5	1	1	AF
A6	1	1	B0
A7	1	1	B1
A8	1	1	B2
A9	1	1	B3
AA	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	B4
AB	1	1	B5
AC	1	1	B6
AD	1	1	B7

All blanks are Os in the matrix.

You can alter the colours of the blocks (lines 60 and 61) and you can put your own matrix in the program by using subroutines. e.g. If you change the contents of address 915 to 00 and 91C to 00, the starting matrix will be filled with blanks. If now you replace line 54 (STRA,RO H'0FB4') with BSTA,UN H'0AE0' (3F0AE0) you can write in the matrix in your own subroutine starting at address AE0 and its final instruction RETC,UN (17) will return control to address 929.

You can also make the display change more slowly by altering the branch instruction at line 181 to say BCTA,LT DELAY and writing a delay routine at DELAY finishing with BCTA,UN BEGIN (1F 09 69).

```
LINE ADDR  OBJECT  E SOURCE

0030 0FBA            OBJP   EQU     H'0FBA' PVI OBJ IND ADR
0031 0FC0            VEC    EQU     H'0FC0' OBJ  IND ADR
0032 0FBC            OBJC   EQU     H'0FBC' MAT IND ADR
0033 0FC2            SQAD   EQU     H'0FC2' SEQ IND ADR
0034 0FBE            FLAG   EQU     H'0FBE' FLAG ADR
0035 0000                   ORG     H'0900'
0036                 ******** START OF PROGRAMME ********
0037 0900 1F0904     PROG   BCTA,UN START
0038 0903 17                RETC,UN
0039 0904 75FF       START  CPSL    H'FF'
0040 0906 20                EORZ    R0
0041 0907 C3                STRZ    R3
0042 0908 CF4E00     CLR1   STRA,R0 H'0E00',R3,-
0043 090B 5B7B              BRNR,R3 CLR1    CLRS MEM.
0044 090D CF4F00     CLR2   STRA,R0 H'0F00',R3,-
0045 0910 5B7B              BRNR,R3 CLR2
0046 0912 0728              LODI,R3 H'28'
0047 0914 0410              LODI,R0 H'10'
0048 0916 CF4F90     FILL   STRA,R0 H'0F90',R3,-
0049 0919 5B7B              BRNR,R3 FILL    MATRIX=10
0050 091B 04FF              LODI,R0 H'FF'
0051 091D CC0F93            STRA,R0 H'0F93' MAT. FF
0052 0920 CC0FAA            STRA,R0 H'0FAA'
0053 0923 CC0F9D            STRA,R0 H'0F9D'
0054 0926 CC0FB4            STRA,R0 H'0FB4'
0055                 ***** BREAKPOINT=929 TO CHANGE MATRIX *****
0056 0929 20                EORZ    R0
0057 092A 0760              LODI,R3 H'60'   CLR SCREEN
0058 092C CF5F00     CLR3   STRA,R0 H'1F00',R3,-
0059 092F 5B7B              BRNR,R3 CLR3
0060 0931 CC1FC1            STRA,R0 H'1FC1' COLOUR
0061 0934 CC1FC2            STRA,R0 H'1FC2' COLOUR
0062 0937 041F              LODI,R0 H'1F'
0063 0939 CC0FBA            STRA,R0 OBJP
0064 093C 0420              LODI,R0 H'20'
0065 093E CC1F0A            STRA,R0 H'1F0A' HC1
0066 0941 CC1F2A            STRA,R0 H'1F2A' HC3
0067 0944 0460              LODI,R0 H'60'
0068 0946 CC1F1A            STRA,R0 H'1F1A' HC2
0069 0949 CC1F4A            STRA,R0 H'1F4A' HC4
0070 094C 0430              LODI,R0 H'30'
0071 094E CC1F0C            STRA,R0 H'1F0C' VC1
0072 0951 CC1F1C            STRA,R0 H'1F1C' VC2
0073 0954 0480              LODI,R0 H'80'
0074 0956 CC1F2C            STRA,R0 H'1F2C' VC3
0075 0959 CC1F4C            STRA,R0 H'1F4C' VC4
0076 095C 04FF              LODI,R0 H'FF'
0077 095E CC1F0B            STRA,R0 H'1F0B' DUP.
0078 0961 CC1F1B            STRA,R0 H'1F1B' HORIZ.
0079 0964 CC1F2B            STRA,R0 H'1F2B' CO-OR.
0080 0967 CC1F4B            STRA,R0 H'1F4B' =FF
0081 096A CC1FC0            STRA,R0 H'1FC0' SIZE
0082 096D 050E       BEGIN  LODI,R1 H'0E'   PGM. LOOP
0083 096F 04B5              LODI,R0 H'B5'
0084 0971 CD0FC0            STRA,R1 VEC
```

```
LINE ADDR  OBJECT  E SOURCE

0085 0974 CC0FC1            STRA,R0 VEC+1
0086 0977 050F              LODI,R1 H'0F'
0087 0979 0490              LODI,R0 H'90'
0088 097B CC0FBD            STRA,R0 OBJC+1
0089 097E CD0FBC            STRA,R1 OBJC
0090 0981 20                EORZ    R0
0091 0982 CC0FBB            STRA,R0 OBJP+1
0092 0985 3F0AAE            BSTA,UN ALTR    OBJECT 1
0093 0988 04BD              LODI,R0 H'BD'
0094 098A CC0FC1            STRA,R0 VEC+1
0095 098D 049A              LODI,R0 H'9A'
0096 098F CC0FBD            STRA,R0 OBJC+1
0097 0992 0410              LODI,R0 H'10'
0098 0994 CC0FBB            STRA,R0 OBJP+1
0099 0997 3F0AAE            BSTA,UN ALTR    OBJECT 2
0100 099A 040F              LODI,R0 H'0F'
0101 099C CC0FC0            STRA,R0 VEC
0102 099F 0469              LODI,R0 H'69'
0103 09A1 CC0FC1            STRA,R0 VEC+1
0104 09A4 04A4              LODI,R0 H'A4'
0105 09A6 CC0FBD            STRA,R0 OBJC+1
0106 09A9 0420              LODI,R0 H'20'
0107 09AB CC0FBB            STRA,R0 OBJP+1
0108 09AE 3F0AAE            BSTA,UN ALTR    OBJECT 3
0109 09B1 040F              LODI,R0 H'0F'
0110 09B3 CC0FC0            STRA,R0 VEC
0111 09B6 0471              LODI,R0 H'71'
0112 09B8 CC0FC1            STRA,R0 VEC+1
0113 09BB 04AE              LODI,R0 H'AE'
0114 09BD CC0FBD            STRA,R0 OBJC+1
0115 09C0 0440              LODI,R0 H'40'
0116 09C2 CC0FBB            STRA,R0 OBJP+1
0117 09C5 3F0AAE            BSTA,UN ALTR    OBJECT 4
0118                  ****REWRITE COMPLETE.****
0119 09C8 0528              LODI,R1 H'28'
0120 09CA 0490              LODI,R0 H'90'
0121 09CC CC0FBD            STRA,R0 OBJC+1  INITIALISE OBJ.
0122 09CF 20                EORZ    R0
0123 09D0 CDCFBC      LOOP  STRA,R0 *OBJC,R1,-
0124 09D3 597B              BRNR,R1 LOOP    CLR MATRIX
0125 09D5 050E              LODI,R1 H'0E'   LOAD SEQ ADR.
0126 09D7 0400              LODI,R0 0
0127 09D9 CD0FC2            STRA,R1 SQAD
0128 09DC CC0FC3            STRA,R0 SQAD+1  (0E00)
0129 09DF 0600              LODI,R2 0       8 BIT CNT.
0130 09E1 CE0FBE            STRA,R2 FLAG    CLR FLAG
0131 09E4 0500        TEST  LODI,R1 0       TESTS 1 BIT
0132 09E6 07FF              LODI,R3 H'FF'     AT SQAD
0133 09E8 0FAFC2            LODA,R0 *SQAD,R3,+
0134 09EB 3E0ADE            BSTA,N  CNT1
0135 09EE 0FAFC2            LODA,R0 *SQAD,R3,+
0136 09F1 3E0ADE            BSTA,N  CNT1
0137 09F4 0FAFC2            LODA,R0 *SQAD,R3,+
0138 09F7 3E0ADE            BSTA,N  CNT1
0139 09FA 0723              LODI,R3 35
0140 09FC 0FAFC2            LODA,R0 *SQAD,R3,+
```

```
LINE ADDR  OBJECT  E SOURCE

0141 09FF 3E0ADE            BSTA,N  CNT1
0142 0A02 0FAFC2            LODA,R0 *SQAD,R3,+
0143 0A05 3E0ADE            BSTA,N  CNT1
0144 0A08 0FAFC2            LODA,R0 *SQAD,R3,+
0145 0A0B 3E0ADE            BSTA,N  CNT1
0146 0A0E 0712              LODI,R3 18
0147 0A10 0FEFC2            LODA,R0 *SQAD,R3
0148 0A13 3E0ADE            BSTA,N  CNT1
0149 0A16 0714              LODI,R3 20
0150 0A18 0FEFC2            LODA,R0 *SQAD,R3
0151 0A1B 3E0ADE            BSTA,N  CNT1
0152 0A1E E502              COMI,R1 H'02'   R1 HOLDS COUNT.
0153 0A20 1806              BCTR,EQ SE0     MAKE =
0154 0A22 E503              COMI,R1 H'03'
0155 0A24 1809              BCTR,EQ SE1     MAKE 1
0156 0A26 180B              BCTR,UN SE0     WRITE NEW BIT
0157 0A28 0713      SE0     LODI,R3 19
0158 0A2A 0FEFC2            LODA,R0 *SQAD,R3
0159 0A2D 9A04              BCFR,N  SE0
0160 0A2F 0401      SE1     LODI,R0 H'01'
0161 0A31 1B01              BCTR,UN WNB
0162 0A33 20        SE0     EORZ    R0      R0 HOLDS NEW BIT
0163 0A34 8C8FBC    WNB     ADDA,R0 *OBJC   WRITE NEW BIT
0164 0A37 8601              ADDI,R2 1       IN MATRIX
0165 0A39 E608              COMI,R2 8
0166 0A3B 181B              BCTR,EQ ENBY    END OF BYTE
0167 0A3D D0                RRL,R0
0168 0A3E CC8FBC            STRA,R0 +OBJC
0169 0A41 0401              LODI,R0 1
0170 0A43 0500              LODI,R1 0
0171 0A45 8C0FC3            ADDA,R0 SQAD+1  2 BYTE ADD
0172 0A48 7708              PPSL    WC      TO SQAD
0173 0A4A 8D0FC2            ADDA,R1 SQAD
0174 0A4D 7508              CPSL    WC
0175 0A4F CC0FC3            STRA,R0 SQAD+1
0176 0A52 CD0FC2            STRA,R1 SQAD
0177 0A55 1F09E4            BCTA,UN TEST
0178 0A58 CC8FBC    ENBY    STRA,R0 +OBJC   1 BYTE STRD.
0179 0A5B 04B6              LODI,R0 H'B6'
0180 0A5D EC0FBD            COMA,R0 OBJC+1
0181 0A60 1E096D            BCTA,LT BEGIN
0182 0A63 20                EORZ    R0
0183 0A64 EC0FBE            COMA,R0 FLAG
0184 0A67 181F              BCTR,EQ SETF
0185 0A69 CC0FBE            STRA,R0 FLAG
0186 0A6C 04A3              LODI,R0 H'A3'
0187 0A6E EC0FBD            COMA,R0 OBJC+1
0188 0A71 180A              BCTR,EQ CHGE
0189 0A73 0C0FBD            LODA,R0 OBJC+1
0190 0A76 A409              SUBI,R0 9
0191 0A78 CC0FBD            STRA,R0 OBJC+1
0192 0A7B 1B05              BCTR,UN CHZZ
0193 0A7D 04A4      CHGE    LODI,R0 H'A4'
0194 0A7F CC0FBD            STRA,R0 OBJC+1
0195 0A82 0403      CHZZ    LODI,R0 3
0196 0A84 0500              LODI,R1 0
```

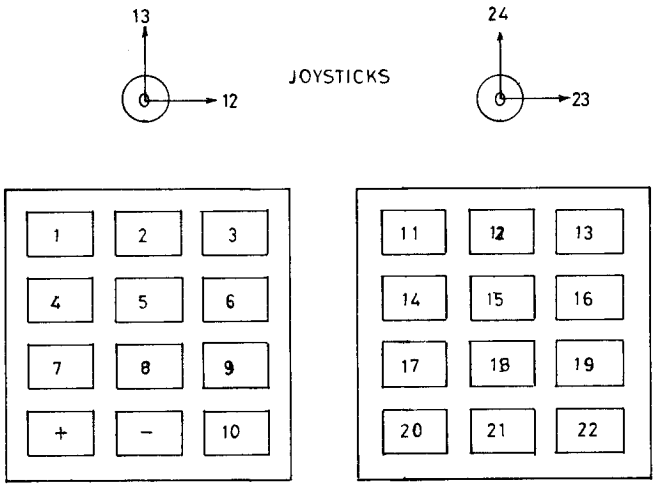
```
LINE ADDR  OBJECT  E SOURCE

0197 0A86 1B11             BCTR,UN OUT1
0198 0A88 0401       SETF  LODI,R0 1
0199 0A8A CC0FBE           STRA,R0 FLAG
0200 0A8D 040A             LODI,R0 H'0A'
0201 0A8F 8C0FBD           ADDA,R0 OBJC+1
0202 0A92 CC0FBD           STRA,R0 OBJC+1
0203 0A95 0401             LODI,R0 1
0204 0A97 0500             LODI,R1 0
0205 0A99 8C0FC3     OUT1  ADDA,R0 SOAD+1  2 BYTE ADD
0206 0A9C 7708             PPSL    WC
0207 0A9E 8D0FC2           ADDA,R1 SOAD
0208 0AA1 7508             CPSL    WC
0209 0AA3 CC0FC3           STRA,R0 SOAD+1
0210 0AA6 CD0FC2           STRA,R1 SOAD
0211 0AA9 0600             LODI,R2 0
0212 0AAB 1F09E4           BCTA,UN TEST
0213                 **** SUBROUTINE ALTR ****
0214 0AAE 070A       ALTR  LODI,R3 H'0A'   TEN BYTE COUNT
0215 0AB0 0FCFBC     WR    LODA,R0 *OBJC,R3,-      MATRIX READ
0216 0AB3 CFEFBA           STRA,R0 *OBJP,R3        PVI OBJ WRITE
0217 0AB6 C1               STRZ    R1
0218 0AB7 0608             LODI,R2 H'08'
0219 0AB9 51         AGN   RRR,R1          ROTATE
0220 0ABA 9A04             BCFR,N  ZER     TEST BIT 7
0221 0ABC 0480             LODI,R0 H'80'
0222 0ABE 1E01             BCTR,UN WRIT
0223 0AC0 20         ZER   EORZ    R0
0224 0AC1 CECFC0     WRIT  STRA,R0 *VEC,R2,-       WRITE BIT 7 (1s)
0225 0AC4 5A7C             BRNR,R2 AGN
0226 0AC6 0C0FC1           LODA,R0 VEC+1
0227 0AC9 0D0FC0           LODA,R1 VEC
0228 0ACC A412             SUBI,R0 H'12'   2 BYTE SUB.
0229 0ACE 7708             PPSL    WC      FROM VEC.
0230 0AD0 A500             SUBI,R1 0
0231 0AD2 7508             CPSL    WC
0232 0AD4 CC0FC1           STRA,R0 VEC+1
0233 0AD7 CD0FC0           STRA,R1 VEC
0234 0ADA 5F0AB0           BRNA,R3 WR
0235 0ADD 17               RETC,UN
0236                 **** SUBROUTINE CNT1 ****
0237 0ADE 8501       CNT1  ADDI,R1 1       ADD 1 TO R1
0238 0AE0 17               RETC,UN
0239                 ****------------------------****
0240 0900                  END     H'0900'

 TOTAL ASSEMBLY ERRORS = 0000
```

7.5. Example 5 - P.V.I. Art

This program shows the capability of the P.V.I. chip. Each key on the keypads and console and each direction of each joystick is used to control one of the P.V.I. facilities, cycling through the possible values of one parameter (which may be one or more P.V.I. registers). When, as a result of pressing a key, a register has been cycled, pressing the keys marked + and - in the diagram will again cycle the register, but the cycle may be stopped at any value. The + key increments the register and the - key decrements it.

The functions are as follows, using the identifying number in the diagram:-

		P.V.I. Address
1.	Left side vertical bars of background	1F80
2.	Right side vertical bars of background	1FA7
3.	Horizontal bar extensions of background	1FA8 to 1FAC
4.	Background and screen colour	1FC6
5.	Score digits	1FC8, 1FC9
6.	Colour of objects 1 and 2	1FC1
7.	Sound frequency	1FC7
8.	Score format	1FC3
9.	Colour of objects 3 and 4	1FC2
10.	Size of four objects	1FC0
11,12,13.	Shape, horizontal co-ord and vertical coord of object 1	1F00 to 1F0C
14,15,16.	Shape, horizontal coord and vertical coord of object 2	1F10 to 1F1C
17,18,19.	Shape, horizontal coord and vertical coord of object 3	1F20 to 1F2C
20,21,22.	Shape, horizontal coord and vertical coord of object 4	1F40 to 1F4C
23.	Horizontal coordinate of duplicate of object 1	1F0B
24.	Separation of duplicates of object 1	1F0D


13
12
JOYSTICKS
24
23
1
2
3
4
5
6
7
8
9
+
−
10
11
12
13
14
15
16
17
18
19
20
21
22
LEFT KEYPAD
RIGHT KEYPAD

```
LINE ADDR  OBJECT  E SOURCE

0034 1F00            PVI    EQU     H'1F00'
0035 1E88            KEYB   EQU     H'1E88' KEYBOARD MATRIX
0036 0005            DELAY  EQU     5
0037 0000                   ORG     H'900'  BEGIN ADDRESS OF RAM
0038 0900 1F09AA     START  BCTA,UN INIT
0039 0903                   ORG     H'903'  INTERRUPT ADDRESS
0040 0903 12         INTRPT SPSU
0041 0904 35                RETE,P  ACCEPT ONLY VRES
0042                 *
0043 0906            FLASH  EQU     $+1     COUNTDOWN BYTE OF FLASH MOD
0044 0905 0600              LODI,R2 0       FLASH
0045 0907 180F              BCTR,Z  POTM0
0046 0909 A601              SUBI,R2 1       FIRST FINISH FLASHING
0047 090B CA79              STRR,R2 FLASH
0048 090D 0D0971            LODA,R1 REG
0049 0910 0D6A00            LODA,R0 DUP,R1
0050 0913 82                ADDZ    R2
0051 0914 CD7F00     FLAS   STRA,R0 PVI,R1
0052 0917 37                RETE,UN
0053                 *
0054 0918 0502       POTM0  LODI,R1 2       READ POTMETER VALUES
0055 091A 0602              LODI,R2 2
0056 091C B440              TPSU    FLAG    ODD OR EVEN FRAME
0057 091E 9806              BCFR,A1 POTM1
0058 0920 7440              CPSU    FLAG    DO HORIZ. POTM. NEXT SCAN
0059 0922 0604              LODI,R2 4       DO VERT. POTM. NOW
0060 0924 1B02              BCTR,UN POTM2
0061 0926 7640       POTM1  PPSU    FLAG
0062 0928 0D5FCC     POTM2  LODA,R0 PVI+H'CC',R1,-
0063 092B C3                STRZ    R3
0064 092C 0E4A0A            LODA,R0 DUP+H'A',R2,-   CHANGE POS OF OBJ1
0065 092F E796              COMI,R3 150
0066 0931 1906              BCTR,GT ADD1
0067 0933 E728              COMI,R3 40
0068 0935 190A              BCTR,GT HOLD
0069 0937 A402              SUBI,R0 2
0070 0939 8401       ADD1   ADDI,R0 1
0071 093B CE6A0A            STRA,R0 DUP+H'A',R2
0072 093E CE7F0A            STRA,R0 PVI+H'A',R2
0073 0941 5965       HOLD   BRNR,R1 POTM2
0074                 *
0075 0943 051B       KEY    LODI,R1 27      KEYBOARD SCAN
0076 0945 0607              LODI,R2 7
0077 0947 0704       KB1    LODI,R3 4
0078 0949 0E5E88            LODA,R0 KEYB,R2,-
0079 094C 60         KB2    LODZ    R0
0080 094D 1A09              BCTR,N  KB3
0081 094F D0                RRL,R0
0082 0950 A501              SUBI,R1 1
0083 0952 FB78              BDRR,R3 KB2
0084 0954 5A71              BRNR,R2 KB1
0085 0956 451F              ANDI,R1 H'1F'
0086 0958 E51F       KB3    COMI,R1 H'1F'   KEY DOWN?
0087 095A 9805              BCFR,EQ KEY1
0088 095C 04FB              LODI,R0 256-DELAY
```

```
LINE ADDR  OBJECT  E SOURCE

0089 095E C808        KEY0    STRR,R0 KEYTIM  RESET KEY TIMER
0090 0960 37                  RETE,UN
0091 0961 0D69C7      KEY1    LODA,R0 TBL1,R1 CONSULT TABLE
0092 0964 34                  RETE,Z          RETURN ON KEY NOT USED
0093 0965 191B                BCTR,P  ACT     NEW OR NEXT PVI REG.
0094 0968             KEYTIM  EQU     $+1     MUST BE + OR - KEY
0095 0967 0400                LODI,R0 0       KEYTIM = KEYPAUSE COUNTDOWN
0096 0969 8405                ADDI,R0 DELAY
0097 096B 1A03                BCTR,N  DECR    SERVICE NEW KEY 1ST ONE TIM
0098 096D C879                STRR,R0 KEYTIM  &AFTER  5 SEC ALSO IN SUCCE
0099 096F 35                  RETE,P
0100                  *
0101 0971             REG     EQU     $+1     SAVE SELECTED PVI ADDRESS
0102 0970 0600        DECR    LODI,R2 0
0103 0972 0E6A00              LODA,R0 DUP,R2
0104 0975 5902                BRNR,R1 INCR
0105 0977 A402                SUBI,R0 2       DECREMENT OR
0106 0979 8401        INCR    ADDI,R0 1       INCREMENTSELECTED PVI REG.
0107 097B CE6A00              STRA,R0 DUP,R2
0108 097E CE7F00              STRA,R0 PVI,R2
0109 0981 37                  RETE,UN
0110                  *
0111 0983             INDEX   EQU     $+1     COUNT OF REG.  ACCESSED BY K
0112 0982 0400        ACT     LODI,R0 0       INDEX
0113 0985             PREVK   EQU     $+1     SAVE OF LAST SELECT KEY VAL
0114 0984 E500                COMI,R1 0       IS SELECT KEY SAME?
0115 0986 9807                BCFR,EQ ACT1
0116 0988 8401                ADDI,R0 1       PICK NXT REG IF SAME KEY
0117 098A ED69C7              COMA,R0 TBL1,R1
0118 098D 1A01                BCTR,LT ACT2
0119 098F 20          ACT1    EORZ    R0      ELSE RESET INDEX
0120 0990 C871        ACT2    STRR,R0 INDEX
0121 0992 C971                STRR,R1 PREVK   SAVE LAST SELECT KEY
0122                  *
0123 0994 C2                  STRZ    R2
0124 0995 0D69E3              LODA,R0 TBL2,R1 CALC ADDR OF SELECTED REG.
0125 0998 E503                COMI,R1 3
0126 099A 1804                BCTR,EQ ACT3
0127 099C E507                COMI,R1 7
0128 099E 9801                BCFR,EQ ACT4
0129 09A0 D2          ACT3    RRL,R2  KEY=3 OR 7 MULTIPLY INDEX BY 2
0130 09A1 82          ACT4    ADDZ    R2
0131 09A2 C84D                STRR,R0 REG
0132                  *
0133 09A4 0419                LODI,R0 25      PRESET FLASH COUNT OF SEL.R
0134 09A6 CC0906      FL      STRA,R0 FLASH
0135 09A9 37                  RETE,UN
0136                  *INITIALISATION
0137                  *
0138 09AA 7620        INIT    PPSU    II
0139 09AC 05D0                LODI,R1 H'D0'   PUT TAPE DATA IN PVI REGS.
0140 09AE 0D4A00      IN1     LODA,R0 DUP,R1,-
0141 09B1 CD7F00              STRA,R0 PVI,R1
0142 09B4 5978                BRNR,R1 IN1
0143 09B6 C94B                STRR,R1 INDEX
0144 09B8 CD0968              STRA,R1 KEYTIM
```

```
LINE ADDR  OBJECT  E SOURCE

0145 09BB C9EA              STRR,R1 *FL+1   FLASH
0146 09BD 0404              LODI,R0 4       ENABLE SOUND
0147 09BF CC1E80            STRA,R0 H'1E80'
0148 09C2 50                RRR,R0
0149 09C3 93                LPSL
0150 09C4 92                LPSU
0151 09C5 1B7E              BCTR,UN $       STANDBY LOOP
0152                    *
0153                    *
0154                    *TABLE CONTAINING MAX NO OF PVI REGS(INDEX) ACCESS
0155                    *
0156 09C7 FF010114      TBL1    DATA    -1,1,1,20,-1,1,2,20,1,1,1,5,0,0,0,0
     09CB FF010214
     09CF 01010105
     09D3 00000000
0157 09D7 0A0A0A0A          DATA    10,10,10,10,2,2,2,2,2,2,2,2
     09DB 02020202
     09DF 02020202
0158                    *
0159                    *TABLE CONTAINING ADDRS OF 1ST OF PVI REGS ACCESSE
0160                    *
0161 09E3 BBC7C680      TBL2    DATA    H'BB,C7,C6,80,BB,C3,C8,81,C0,C2,C1,
     09E7 BBC3C881
     09EB C0C2C1A8
     09EF BBBBBBBB
0162 09F3 40201000          DATA    H'40,20,10,00,4A,2A,1A,0A,4C,2C,1C,
     09F7 4A2A1A0A
     09FB 4C2C1C0C
0163                    *
0164 09FF                   ORG     START+H'100'    DUPLICATE OF PVI RE
0165 0A00 FF818181      DUP     DATA    H'FF,81,81,81,81,81,81,81,81,FF,27,
     0A04 81818181
     0A08 81FF2728
     0A0C 140A0000
0166 0A10 FF814224          DATA    H'FF,81,42,24,18,18,24,42,81,FF,2E,
     0A14 18182442
     0A18 81FF2E2F
     0A1C 201D0000
0167 0A20 55AA55AA          DATA    H'55,AA,55,AA,55,AA,55,AA,55,AA,50,
     0A24 55AA55AA
     0A28 55AA5096
     0A2C 14FF0000
0168 0A30 AA55AA55          DATA    H'AA,55,AA,55,AA,55,AA,55,AA,55,50,
     0A34 AA55AA55
     0A38 AA555096
     0A3C 14FF0000
0169 0A40 00000000          DATA    H'00,00,00,00,00,00,00,00,00,00,00,
     0A44 00000000
     0A48 00000000
     0A4C 00000000
0170 0A50 00000000          DATA    H'00,00,00,00,00,00,00,00,00,00,00,
     0A54 00000000
     0A58 00000000
     0A5C 00000000
0171 0A60 00000000          DATA    H'00,00,00,00,00,00,00,00,00,00,00,
     0A64 00000000
```

```
LINE ADDR  OBJECT   E SOURCE

     0A68 00000000
     0A6C 00000000
0172 0A70 00000000            DATA    H'00,00,00,00,00,00,00,00,00,00,00,
     0A74 00000000
     0A78 00000000
     0A7C 00000000
0173 0A80 FFFEFFFF            DATA    H'FF,FE,FF,FF,FF,FE,FF,FF,FF,FE,FF,
     0A84 FFFEFFFF
     0A88 FFFEFFFF
     0A8C FFFEFFFF
0174 0A90 FFFEFFFF            DATA    H'FF,FE,FF,FF,FF,FE,FF,FF,FF,FE,FF,
     0A94 FFFEFFFF
     0A98 FFFEFFFF
     0A9C FFFEFFFF
0175 0AA0 FFFEFFFF            DATA    H'FF,FE,FF,FF,FF,FE,00,00,03,09,09,
     0AA4 FFFE0000
     0AA8 09090989
     0AAC 89000000
0176 0AB0 00000000            DATA    H'00,00,00,00,00,00,00,00,00,00,00,
     0AB4 00000000
     0AB8 00000000
     0ABC 00000000
0177 0AC0 F6341801            DATA    H'F6,34,18,01,00,00,4A,15,00,00,00,
     0AC4 00004A15
     0AC8 00000000
     0ACC 00000000
0178 0900                     END     START

 TOTAL ASSEMBLY ERRORS = 0000
```

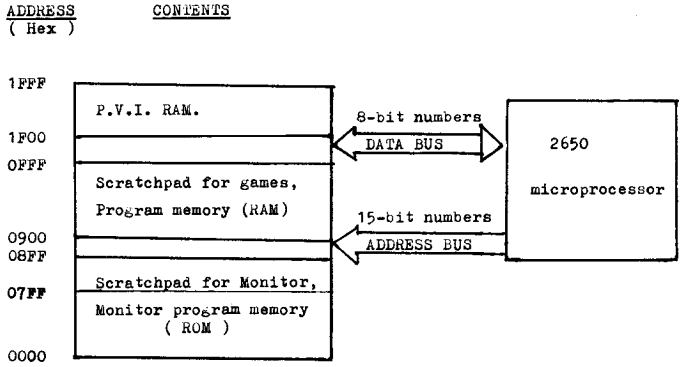


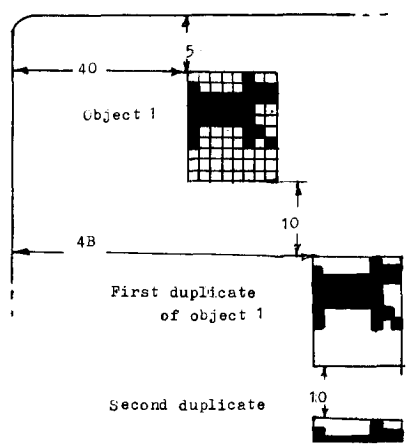
FIGURE 1 - MEMORY CONTENTS.

ADDRESS	CONTENTS		HEX. FORM.
1F00	0000 0100	Object	04
1F11	1000 0111		87
1F12	1111 1111		FF
1F13	1111 1100		FC
1F14	1111 1100		FC
1F15	1000 0110		86
1F16	1000 0101		85
1F17	0000 0000		00
1F18	0000 0000		00
1F19	0000 0000		00
1F1A	Horizontal Co-ord. of Object 1		40
1F1B	Hor. Co-ord of Duplicates		4B
1F1C	Vertical Co-ord. of Object 1		05
1F1D	Vert. Separation of Duplicates		10

FIGURE 2 -

Example of P.V.I Object -

A dog in object 1.



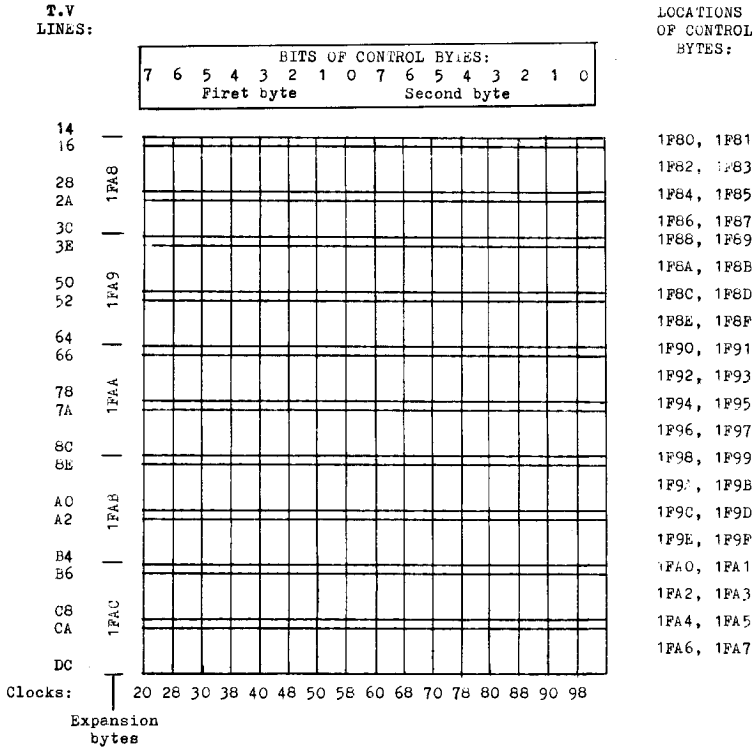


FIGURE 3. - LOCATIONS FOR SPECIFYING THE BACKGROUND.

APPENDIX 1

MONITOR FUNCTIONS

1. MEM: Gives display of memory contents at the address given by Ad = HHHH. (Type address required and '+'.) '+' increments address and '-' decrements it.

   Typing pairs of hex. characters fills locations automatically but the last pair should be followed by '+' to enter it.

2. REG: Gives display of register contents (R0 to R8). '+' increments and '-' decrements register number. Register contents are altered by typing two hex. characters and '+' or '-'.

3. PC: On pressing PC and '+' the monitor will try to run the program from the program counter address. Type in new values and '+' to alter.

4. BP: One or two breakpoints (press BP once/twice) can be set by typing in the address and '+'. When the processor reaches the breakpoint address it jumps back to the monitor program saving all registers and clearing the breakpoint. The program can be restarted from the breakpoint by pressing PC and '+' (To restart from any other address you must alter PC).

   If the processor does not get to the breakpoint it will not clear it. This can be done by re-entering the correct program code into the memory at the breakpoint address and the following address (BP+1).

5. RCAS: Reads a program from cassette. If you make FIL = a single digit, the processor will read only that file; if nothing is entered it will read any file. '+' reads the file into memory, '-' verifies the file by comparing it to the memory.

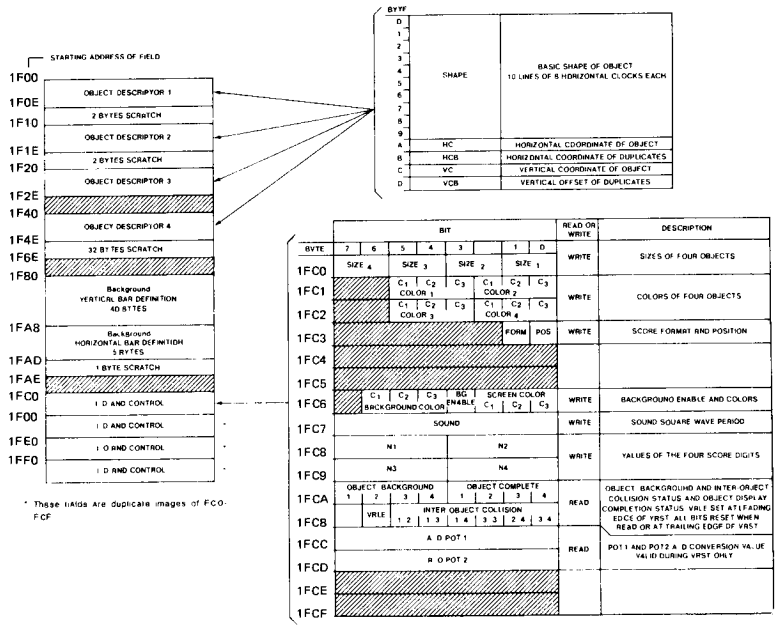
6. WCAS: Writes a program to cassette. You should enter: - Start address, End address and address for the program counter to be set to, and file number. '+' starts writing the file.

7. RESET: After powering up, the system has to be reset manually. RESET forces the processor to begin the monitor program which will display up to 6 lines of status (anything that has previously been written in monitor). RESET can be used to stop user programs or tape read or write. ++++ is displayed on the bottom line, PC is set to 900, breakpoints are not cleared.

8. START: When in monitor, START clears the monitor status and the screen and displays ++++ on the bottom line. PC is set to 900, breakpoints are set to 0.

APPENDIX 2

P.V.I. LOCATIONS



APPENDIX 3

INSTRUCTION CODES

INSTRUCTION : 1ST DIGIT	MODE : 2ND DIGIT
LOD - - - - 0	Z Rn - - - - 0+n
STR - - - - C	I,Rn - - - - 4+n
ADD - - - - 8	R,Rn - - - - 8+n
SUB - - - - A	A,Rn - - - - C+n
COM - - - - E	
AND - - - - 4	
EOR - - - - 2	
IOR - - - - 6	
BRN - - - - 5	R,Rn - - - - 8+n
BSN - - - - 7	A,Rn - - - - C+n
BIR - - - - D	
BDR - - - - F	
BCT - - - - 1	R,con - - - - 8+con
BST - - - - 3	A,con - - - - C+con
BCF - - - - 9	
BSF - - - - B	
RETC,con - - - - 14+con	n is Register number
RETE,con - - - - 34+con	n = 0 for R0
RRR,Rn - - - - 50+n	= 1 for R1
RRL,Rn - - - - D0+n	= 2 for R2
TMI,Rn - - - - F4+n	= 3 for R3
DAR,Rn - - - - 94+n	con is Branch condition
CPSL - - - - 75	con = 0 for Z and EQ
PPSL - - - - 77	= 1 for P and GT
TPSL - - - - B5	= 2 for N and LT
CPSU - - - - 74	= 3 for UN
PPSU - - - - 76	After TMI only
TPSU - - - - B4	con = 0 if all 1s,2 if not

INDEXING : n is Index Register
Add to 1st DIGIT of address (3rd CODE DIGIT)
INDEX, + = 2 : INDEX, - = 4 : INDEX only = 6

On Absolute addressing
R0 only (Not BRANCHES)

INDIRECT addressing : Any R or A mode.
Add 8 to 1st DIGIT of address (3rd CODE DIGIT)

C/P/TPSL : C = 01, COM = 02, OVF = 04, WC = 08,
RS = 10, IDC = 20, CC0 = 40, CC1 = 80
C/P/TPSU : Flag(joystick forward/lateral) = 40, II = 20, VRLE = 80

APPENDIX 4

GLOSSARY

ASSEMBLE	-	To translate a program from mnemonic (assembly) code to object code.
ASSEMBLY CODE	-	A set of mnemonics representing microprocessor instructions.
BINARY CODE	-	A system of numbering using only the digits 0 and 1.
BINARY DIGIT ('BIT')	-	A digit in binary code.
BRANCH (OR JUMP)	-	An instruction causing the program to take its next instruction from a memory location other than the next one.
BREAKPOINT	-	An address at which the program is temporarily stopped and return to monitor mode.
BYTE	-	A set of eight bits, often representing a binary number.
FLAG	-	A single bit, used to record an event, decision etc.
HEXADECIMAL	-	A system of arithmetic with a base of 16.
INTERFACE	-	A circuit to change the form of a signal, to make it acceptable to another device.
LOCATION	-	A position in the memory which can hold one byte.
LOOP	-	A set of program instructions which are automatically repeated.

Contd/...

MEM - A keypad button used to examine or change the contents of a memory location.

MONITOR - The program used to supervise the microprocessor function.

MODULE - An electronic circuit which can be plugged into, or connected to, other modules to form a complete system.

NIBBLE - Four bits, or half a byte.

OBJECT - (In connection with the PVI) a shape which can be held in memory and automatically written to the television screen.

OBJECT CODE - A numerical code in which the program is sent to the microprocessor.

PROGRAM COUNTER (PC) - A register in which the address of the current instruction is held.

PROGRAMMABLE VIDEO INTERFACE - An integrated circuit used to produce the television picture, hold objects, etc.

RANDOM ACCESS MEMORY (RAM) - Memory in which program, variables, etc. can be stored temporarily.

REG - A keypad button used to examine or alter contents of registers.

REGISTER - A circuit capable of containing a byte.

READ ONLY MEMORY (ROM) - Permanent memory, used to contain the monitor program.

Contd/...

VIDEO	-	The signal from the games unit to the display unit to produce the picture.
VERTICAL RESET LEADING EDGE (VRLE)	-	A synchronising signal permitting the microprocessor to take action between frames of the television picture.

APPENDIX 5

MONITOR SUBROUTINES SUMMARY

1. KEYSCN : Keyboard scan routine.

   Entry : 014E

   : Call during vertical reset interrupt routine.

   Input : MSB set to 0 in MKBST(089F) or RKBST (089E) or LKBST(089D)

   Registers used : R0, R1, R2, R3 bank 0.

   Subroutine levels used : 2.

   Output : MKBST (089F) . Joined keyboard

   LKBST (089D) . Left keyboard

   RKBST (089E) . Right keyboard

2. CLRSCR : Clear screen routine

   Entry : 0575

   Registers used : R0, R2 bank 0.

   Subroutine levels used : 1.

   Output : P.V.I. objects cleared

3. CONVRT : Line to image conversion routine.

   Entry : 01D0

   Input : 0890....0897 - eight character codes.

   Registers used : R0, R1, R2, R3 bank 0.

   Subroutine levels used : 2.

   Output : Character images in monitor scratch displayable via subroutine OUTPUT.

4. OUTPUT : Display of characters image on screen.

   Entry : 0057 (monitor settings) or 006F (user defined)

   : Call in vertical reset interrupt routine.

   Input : Character images in monitor scratch-pad. (800 - 88F)

   Registers used : R0, R1, R2 bank 0.

   Subroutine levels used : 1.

5. SCROLL : Scroll up of character images one line position.

   Entry : 028A (6 lines) or 028C (1-5 lines).

   Input : R1 = 0, E8, D0, B8, A0 for 1-5 lines.

   Registers used : R0, R1, R2 bank 0.

   Subroutine levels used : 1.

APPENDIX 6

CONVERSION TABLES

(a) Hexadecimal to Decimal

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
1	1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
2	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
3	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
5	5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
6	6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
7	7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
8	8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
9	9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
A	10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
B	11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
C	12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
D	13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
E	14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
F	15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255

(b) Decimal to Hexadecimal

	0	10	20	30	40	50	60	70	80	90
0	0	A	14	1E	28	32	3C	46	50	5A
1	1	B	15	1F	29	33	3D	47	51	5B
2	2	C	16	20	2A	34	3E	48	52	5C
3	3	D	17	21	2B	35	3F	49	53	5D
4	4	E	18	22	2C	36	40	4A	54	5E
5	5	F	19	23	2D	37	41	4B	55	5F
6	6	10	1A	24	2E	38	42	4C	56	60
7	7	11	1B	25	2F	39	43	4D	57	61
8	8	12	1C	26	30	3A	44	4E	58	62
9	9	13	1D	27	31	3B	45	4F	59	63

DEC.	HEX
100	64
200	C8
300	12C
400	190
500	1F4
600	258
700	2BC
800	320
900	384
1000	3E8

(c) Relative Address Calculations

A relative address consists of six bits and a sign bit, i.e. a range of B'100 0000' to B'011 1111', which is H'40' to H'3F', equivalent in this instance to D'-64' to D'63'. Displacements can be calculated by starting at the byte following the relative address and counting the number of bytes to the desired location (see below for example), either directly in hexadecimal or in decimal followed by conversion using appendix 6A. If the displacement is negative (e.g. jump to an earlier address) counting must either proceed backwards in hexadecimal (0,7F,7E,7D etc) or negatively in decimal with the following conversion:-

DEC	HEX
- 1	7F
- 2	7E
- 3	7D
- 4	7C
- 5	7B
- 6	7A
- 7	79
- 8	78
- 9	77
-10	76
-11	75
-12	74
-13	73
-14	72
-15	71
-16	70

DEC	HEX
-17	6F
-18	6E
-19	6D
-20	6C
-21	6B
-22	6A
-23	69
-24	68
-25	67
-26	66
-27	65
-28	64
-29	63
-30	62
-31	61
-32	60

DEC	HEX
-33	5F
-34	5E
-35	5D
-36	5C
-37	5B
-38	5A
-39	59
-40	58
-41	57
-42	56
-43	55
-44	54
-45	53
-46	52
-47	51
-48	50

DEC	HEX
-49	4F
-50	4E
-51	4D
-52	4C
-53	4B
-54	4A
-55	49
-56	48
-57	47
-58	46
-59	45
-60	44
-61	43
-62	42
-63	41
-64	40

Example:

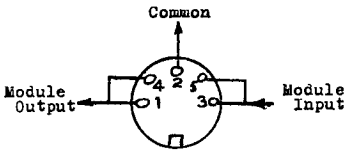
```
950    05 06          LODI,R1  6
952    06 50 LOOP2    LODI,R2  50
954    C0    LOOP1    NOP
955    FA xx          BDRR,R2 LOOP1
957    F9 yy          BDRR,R1 LOOP2
959                   etc.
```

Here xx is from 957 to 954 which is D'-3', or H'7D' from the table and yy is from 959 to 952 which is D'-7' or H'79' from the table

APPENDIX 7

CONNECTING A TAPE RECORDER

A cassette recorder may be connected to the Hobby Module to permit you to store your programs on tape. The cassette recorder is connected to the Hobby Module through a 5-pin, 180 degree DIN socket, with standard pinning as shown below. If your cassette recorder does not have a DIN socket you may connect the Hobby Module DIN socket to the microphone input and speaker output (or earphone) jacks on the cassette recorder, following the same input/output connections as shown below.



5-pin socket on Hobby Module

The output signal from the Hobby Module is 10 millivolts p-p/k-ohm with a maximum of 500mv. This is suitable for both diode and microphone input types.

The Hobby Module requires an input signal of 0.5 to 5 volt p-p. This is obtainable from most cassette recorders, though most hi-fi tape decks are not suitable.

Good quality tapes are recommended. Cheap tapes cause errors due to drop-outs.

Interference filters are built into the Hobby Module but a minimum distance of one meter between the cassette recorder and the television set is recommended.

RECORDING PROCEDURE

The most critical aspect in recording programs with the Hobby Module is to ensure the output level of the cassette recorder falls within the working limits of the Hobby Module. The following procedure may be used to adjust the volume control of your cassette recorder to the correct level.

Press	TV Displays
WCAS (Write to cassette)	bEG
9ØØ	bEG = 900
+	End =
CØØ	End = C00
+	SAd =
9ØØ	SAd = 900
+	FIL =
1	FIL = 1

Now the Hobby Module is ready to write on the tape. Insert a blank cassette and allow to run a few inches.

Press + and recording begins. The TV should show FIL = 1 at the top of the screen. After a successful recording, a whole page of information would be displayed.

To verify the tape we proceed with the following steps:

Press	TV Displays
RCAS (Read cassette)	FIL =
1	FIL = 1
-	FIL - 1 (at top of screen)

Rewind the tape and play back at medium volume level. Two dots under the " - " sign at the top of your TV screen will blink at regular intervals if data is being input correctly from the cassette to the Hobby Module.

If these dots are missing you will have to experiment with higher or lower volume levels. Repeat the verifying process until the blinking dots indicate the correct volume level is being used.

If the screen displays an error or the dots blink at an irregular frequency the volume level should be adjusted and the verification process repeated.

10 REFERENCES & BIBLIOGRAPHY

1. For information on writing flowcharts:

   Problem Solving and Flowcharting Ronald E. Elliott

   Reston Publishing Co. Inc.

2. For information on the Integrated Circuits:

   Data sheets on 2650 microprocessor and 2636 p.v.i. chip

   Mullard Ltd. Torrington Place, London WC1E.7HD

3. For information on the 2636 p.v.i. chip:

   Electronic Components & Applications Vol 1, no. 1,

   S.J. Op Het Veld - Microprocessor Controlled Video Games.

   Mullard Ltd. Torrington Place, London WC1E.7HD

ADDENDUM

1. SETTING VERTICAL OFFSETS

The next duplicate offsets are latched into the PVI at the completion of an object or duplicate. Thus for correct operation the vertical offset of a duplicate must be written to the PVI before the previous duplicate or object is completed on the television screen display.

2. INTERRUPTS

When using interrupts, the following procedure is recommended.

(a) Do not use bank 1 registers in the main programme. (If you do you must store and reload them as with R0).

(b) At the start of the Interrupt routine,

(1) Store R0 in RAM.

(2) Store the Program Status Lower in a special location (see below) in RAM.

(3) Switch register banks.

(c) At the end of the Interrupt routine,

(1) Re-load R0 from RAM.

(2) Clear Program Status Lower.

(3) Reload Program Status Lower.

N.B. The PSL must be stored in the next to the last byte of the Interrupt routine and the previous byte must have 77 written in it. The processor will then restore the PSL correctly as it executes the last three instructions of the Interrupt.

E.G. START INTERRUPT

Addr.	Hex Code	Instruction		Comment
	CC (XX XX)	STRA,R0	XX XX	Into RAM.
	I3	SPSL		Store PSL in R0.
	CC (ZZ+1)	STRA,R0	ZZ+1	(YY) into special location.
	77 10	PPSL	10	Switch register bank.
		Interrupt Routine		
	OC (XX XX)	LODA,R0	XXXX	Restore R0.
	75 FF	CPSL	FF	Clear PSL
ZZ	77 (YY)	PPSL	(YY)	PPSL to Mask (YY).
ZZ+2	37	RETE,UN		Return and Enable.

PROBLEM NO.	START ADDR	PROGRAM
DESCRIPTION		

RADOFIN HOBBY MODULE

MICROPROCESSOR PROGRAMMING SHEET

DATE	SHEET OF

LINE	ADDRS	DATA			LABEL	SYMBOLIC INSTRUCTION		COMMENT
		B0	B1	B2		OPCODE	OPERANDS	
1								
2								
3								
4								
5								
6								
7								
8								
9								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								
21								
22								
23								
24								

9398 9138 0011

©Copyright 1980, Radofin Electronics (Far East) Ltd.
Printed in Hong Kong

IB-2241-1